

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Java

RESTful Web Service 实战

(第2版)

Java RESTful Web Services in Action
Second Edition

韩 陆 著



内容简介

本书系统、深度讲解了如何基于Java标准规范实现REST风格的Web服务，由拥有10余年开发经验的阿里云大数据架构师撰写，第1版上市后广获赞誉，成为该领域的畅销书。第2版对全书进行了优化和重构，不仅根据最新的技术版本对原有过时内容进行了更新，而且还根据整个技术领域的发展增添了新的内容。除此之外，还对第1版中存在的不足进行了优化，使得内容更加与时俱进、更加有价值。不仅深刻解读了最新的JAX-RS标准和其API设计，以及Jersey的使用要点和实现原理，还系统讲解了REST的基本理论，更重要的是从实践角度深度讲解了如何基于Jersey实现完整的、安全的、高性能的REST式的Web服务，书中包含大量示例代码，实战性强。

全书共10章，包括JAX-RS2入门、REST API设计、REST请求处理、REST服务与异步、REST客户端、REST测试、微服务、容器化、JAX-RS调优、REST安全等内容。书中从基础概念开始，结合大量示例和实现代码，将REST理论与Java实现相结合，循序渐进地阐述Java REST式服务，为读者提供更精炼、更准确、更全面的参考。



Java

RESTful Web Service 实战

(第2版)

Java RESTful Web Services in Action
Second Edition

韩陆 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java RESTful Web Service 实战 / 韩陆著. —2 版. —北京: 机械工业出版社, 2016.7
(2016.10 重印)
(Java 核心技术系列)

ISBN 978-7-111-54213-1

I. J… II. 韩… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2016) 第 156331 号

Java RESTful Web Service 实战 (第 2 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 李 艺

责任校对: 董纪丽

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2016 年 10 月第 2 版第 2 次印刷

开 本: 186mm×240mm 1/16

印 张: 18.75

书 号: ISBN 978-7-111-54213-1

定 价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Foreword 第2版序一

韩陆是我在阿里巴巴的同事，业余时间大家经常一起聊新的 Java 技术。REST 对当前软件开发非常重要，除了我们一直了解的 Service API、Open API、移动端对后端的 Gateway API 调用（这些基本都是 REST 模式设计的），现在很多的 DevOps 操作也是通过 REST API 完成的，如我们了解的 Docker 和 SpringBoot Actuator API 都是 REST 风格的，另外 HTTP/2 的逐步采用，也为 REST 带来更多的功能和性能的提升。对 Java 程序员来说，本书非常难得，你可以深入了解 JAX-RS 标准和 Jersey 框架；为了方便落地开发，书中更着重介绍了 Spring Boot 和 Spring Cloud，这些知识目前涉及的中文图书并不多；最后结合 Docker 容器技术，给出了完整基于 SpringBoot REST 服务应用容器部署的思路。本书的每一个技术点都可以单独成书，用以详细阐述，但是能够浓缩到一本图书中，挑战和难度确实比较大，希望这些新的技术和思想能够帮助到真正前进的程序员。

阿里巴巴资深技术专家、速卖通中间件掌门人 陈立兵（花名：雷卷）

第2版序二 *Foreword*

认识韩陆已有近10年的时间，那时他在北京航空航天大学软件学院做硕士毕业论文。他的论文写的是面向对象数据库引擎的设计与实现，完全自主实现了包括文件读写、缓存、索引和事务、数据访问接口等完整的面向对象数据库引擎。从那时就了解到他是一位技术达人，喜欢钻研和实践各种最新的技术。所以当听说他出版本书时一点都不觉得意外，他就是这样一个热衷于新技术的人。

早在本书第1版的时候，他就找到我希望为本书写一篇序，那时我婉拒了，因为我本人对RESTful相关技术并没有太多的接触，不敢贸然推荐。作为一种轻量级Web服务实现架构，两年多来RESTful架构得到了普遍认可和使用；越来越多的学生也开始学习相关的技术，而本书就是这方面非常有意义的参考资料。书中首先系统地解读了JAX-RS2标准，之后基于JAX-RS标准的参考实现：Jersey开发框架，系统地讲解了如何基于该框架开展RESTful Web服务的实践。本书实践性很强，体系较为完整，涵盖了RESTful Web服务开发各个层面的问题；书中不仅提供了一些典型场景的代码示例，还有完整的项目案例的讲解，这些实践代码能够有助于读者开展具体的项目实践。与第1版相比，第2版还新增了有关微服务和容器化等目前热门应用技术实践的内容，有助于读者了解最新的技术发展方向。

北京航空航天大学 谭火彬

Foreword 第1版序一

——REST 开发的理想与现实

REST 是一种分布式应用的架构风格，也是一种大流量分布式应用的设计方法论。REST 是由（构成了 Web 基础架构的）HTTP、URI 等规范的主要设计者 Roy Fielding 博士在其 2000 年的博士论文（中文版名为《架构风格与基于网络应用程序的架构设计》）中提出的。到目前为止，关于 REST 最系统、最全面的论述，仍然是 Fielding 的博士论文。

REST 就是 Web（World Wide Web，简称 Web 或者 WWW）本身的架构风格，是设计、开发 Web 相关规范、Web 应用、Web 服务的指导原则。不符合 REST 风格要求的架构和技术，很难在 Web 这个生态系统中得到繁荣发展。在我看来，Roy Fielding 博士就是 15 年以来对于分布式应用架构设计理论贡献最大的人。Fielding 在 HTTP 规范的设计过程中，并没有采用当时大行其道的 DO（Distributed Object，分布式对象）风格，而是自出机杼、另辟蹊径，提出了一整套新的设计方法论。Fielding 的开创性工作，极大地推动了分布式应用设计理论的发展。

有趣的是，其实基于 SOAP/WSDL 的“大 Web Service”（以下简称 Web Service），几乎是与 REST 同时发展起来的。虽然在 Web Service 中也使用了对象，但是 Web Service 其实是 RPC 风格的，而不是 DO 风格的。Web Service 在最初几年发展很快，很大原因是它解决了 DO 风格难以解决的异构系统（不同的硬件系统、不同操作系统、不同的编程语言，等等）之间互操作性的问题。

然而遗憾的是，设计 Web Service 协议栈的核心人员，几乎都是来自于企业应用阵营的，尤其是来自于 IBM 和微软两家公司的人。这些企业应用的专家们没有充分认识到 Web 基础架构的巨大优点，甚至可以说并没有理解 HTTP 协议究竟是用来做什么的、为何要如此设计。在 Web Service 协议栈的设计之中，仍然有深深的企业应用痕迹。Web Service 虽然

宣称能够很好地支持互操作，然而因为协议栈的复杂性很高，在实战中互操作性并不好（例如升级过程困难而且复杂）。此外，Web Service 仅仅将 HTTP 协议当做一种传输协议来使用，还依赖 XML 这种冗余度很高的文本格式，这导致 Web Service 应用性能低下。很多开发团队宁可使用 Hessian 等轻量级的 RPC 协议，也不愿意使用 Web Service。在面向互联网的大流量 Web 应用（包括 Web 服务在内）这种运行环境中，Web Service 在复杂性、互操作性、性能、可伸缩性等方面的短板更加突出。因此，设计今日面向互联网的 API，已经很少有人会考虑 Web Service。这使得 Web Service 的使用被局限在企业应用运行环境之中，其名称中的“Web”更像是一个笑话（除了都使用 HTTP 协议，基本上与 Web 没什么关系）。假如在 2000 年，设计 Web Service 规范的专家们，能够认真读一下 Fielding 的博士论文，或者找 HTTP、URI 等 Web 基础架构规范的核心设计人员深入交流一下，Web Service 很可能就不是现在这个样子了。不过，历史是无法假设的。

在 Java 世界中，与大 Web Service 相对应的规范是 JAX-WS。在大 Web Service 已经成为明日黄花之后，Java 世界急需一套新的规范来取代 JAX-WS。这套新的规范就是 JAX-RS: Java 世界开发 RESTful Web Service（与 RESTful API 含义相同，可混用）的规范。虽然起步很晚，毕竟走上了正确的道路。

从 Java EE 6 开始，JAX-RS 在 Java EE 版图中，作为最重要的组成部分之一，逐步取代了 JAX-WS 的地位。在所有 Java EE 相关规范中，JAX-RS 是优点很突出的一个。例如，完全基于 POJO、很容易做单元测试、将 HTTP 作为一种应用协议而不是可替代的传输协议（因此提高了性能）、优秀的 IDE 集成，等等。可以说，在大多数场合，JAX-RS 完全可以取代 JAX-WS，作为 Java Web Service 开发的主要技术。JAX-RS 同样也可以完全取代 Hessian 等基于 HTTP 协议的 RPC 风格远程调用协议。毕竟 HTTP 本身就是一种 REST 风格的应用协议，以 REST 风格来使用 HTTP，才是最高效的使用方式。

Jersey、CXF 等支持 JAX-RS 规范的 REST 开发框架还支持输出 WADL。WADL 支持客户端代码自动生成，还可以将 WADL 导入到 SoapUI 等测试工具中，然后做自动化集成测试。从开发 Java 企业应用、取代 JAX-WS 的角度来看，JAX-RS 已经做得非常棒了。

尽管如此，不可不提的是，JAX-RS 这套规范，仍然存在着很多遗憾。需要特别指出的是，JAX-RS 规范并不等于 REST 架构风格本身，REST 的内涵要比 JAX-RS 广泛得多。学会了使用 JAX-RS 了，并不等于就完全理解了 REST，开发者仍然需要下工夫认真学习一下本源的 REST 究竟是什么。

例如，JAX-RS 规范对于应该如何定义一个资源，以及应该如何使用 HTTP 作为一个统一接口来操作资源，显然缺乏必要的指导。有很多开发者只是简单地将以前 JAX-WS 中的一个 endpoint 接口转换成一个资源接口。接口的方法太多，导致映射到的 HTTP 方法不够

用，这也难不倒他们，在 URI 路径中加一些字符串就能够解决（例如，三个接口方法都映射到 POST，但是其 PATH 不同）。这样的开发方式非常常见，虽然开发者使用了 JAX-RS 规范，但是开发方式完全是 RPC 风格的，可以说与 REST 风格没有任何关系。

此外，JAX-RS 规范目前尚不支持 HATEOAS（将超文本作为应用状态的引擎，REST 风格的核心特征之一），从著名的 Richardson 成熟度模型（由《RESTful Web APIs》的作者 Richardson 提出）来衡量，基于 JAX-RS 规范实现的 RESTful API 仅仅能够达到成熟度模型的第二级，即支持资源抽象、统一接口的“CRUD 式 Web 服务”。

可以这样说，JAX-RS 规范与真正的 REST 风格，覆盖的范围其实是不同的。JAX-RS 覆盖的是简单基于 HTTP 协议（没有使用 SOAP/WSDL）的各种远程调用需求，很多需求对于可伸缩性、松耦合的要求并不高，仅仅是希望使用 HTTP 本身来取代大 Web Service 作为一种轻量级、容易测试的远程调用协议。REST 架构风格的严格要求，在这些场合并不是非常重要。慵懒是人类的天性，大多数开发者写代码只是为了解决手头的问题，JAX-WS 并不好用，JAX-RS 解救了他们。

如果按照 Roy Fielding 博士的严格要求（REST APIs must be hyper-text driven），那么包括 JAX-RS 规范在内都不能算是真正的 RESTful。然而，从实战角度，我认为革命不分先后，只要能够达到 Richardson 成熟度模型第一级，即有清晰的资源抽象，就可以认为是 RESTful API 了。如果连第一级都达不到，所设计的架构根本就不是面向资源的，那八成还是 RPC 风格的，就没有必要非要往 RESTful API 阵营里面挤了。从来没有人说过 RPC 就是万恶的，RPC 在企业应用的大多数场合其实都非常有效，只是不适合面向互联网的大流量 Web 应用而已。

因此，能够完美支持 HATEOAS，攀登到成熟度模型第三级，是一种理想情况（当然也是值得追求的）。而通过部分拥抱 REST 风格的要求，来更好地解决手头的问题，是更多开发者所面对的现实情况。JAX-RS 反映的正是这种现实情况，从实战的角度，它是一套非常有用也很好用的规范。

韩陆兄的新著《Java RESTful Web Service 实战》是 JAX-RS 规范方面的专著，也是国内第一本 REST 开发的原创著作。这本书的实战性非常强，全面介绍了 JAX-RS 2.0 的方方面面，可以作为一线 Java 分布式应用开发者的案头必备书。如同我在前面所指出的，JAX-RS 规范并不等于 REST 架构风格本身，它们有着不同的覆盖范围。在本书中，作者也介绍了很多设计 RESTful API 的最佳实践，这些内容假如读者不理解 REST，甚至在亲自阅读了 JAX-RS 规范之后也未必能够总结出来。读者在阅读本书的过程中，不应该仅仅满足于掌握了 JAX-RS 开发的基本方法、解决了手头的问题、用其完全取代 JAX-WS，更重要的是，读者还应该就 REST 架构风格本身做更多的学习。幸运的是，除了本书之外，目前 REST 设计

和开发方面的图书资料已经非常多了。

本书的内容非常严谨，有非常好的系统性，对于设计开发大流量 Web 服务会面临的各种问题都有涉及。特别是在自动化测试方面着墨颇多，在我看来是本书的一大亮点。RESTful API 的自动化测试非常重要，需要在设计之初就充分考虑到。本书是一本难得的原创佳作，值得所有 Java 分布式应用的开发者购买。

理想富丽丰满，现实贫瘠骨感，追求理想和注重解决现实问题其实并不矛盾。JAX-RS 规范的发展，反映出了 Java 社区在更好地开发 RESTful Web Service 方面的求索。尽管存在争议，在我看来，规范化是推动 RESTful Web Service 取得更大发展的必由之路。目前对于优秀的 RESTful API 有哪些判断标准，Web 开发者社区已经达成了高度共识，也积累了大量非常有价值的成果。JAX-RS 规范的发展，离不开 Web 开发者社区的这些成果。在未来的 JAX-RS 3.0 规范中，我们将会看到更多令人兴奋的成果被规范化。JAX-RS 2.0 已经做得不错了，但是在 RESTful Web Service 规范化的道路上，其实才刚刚起步，任重而道远。

李锐 于上海

Foreword 第1版序二

半年前初识韩陆的时候，我们就聊到他正在写的这本书，当得知我从2006年就参与了Apache CXF开发，他立即邀请我为他的新书写序，我也就欣然答应了。

Apache CXF作为JAXWS以及JAX-RS规范的实现框架，已经成为很多Web服务开发者必选的开发框架。作为这一框架的开发维护者之一，我的日常工作经常需要熟悉这些JSR规范，并实现JSR所定义的API，解决最终用户的使用问题。

熟悉Java的人大多都听说过JSR (Java Specification Requests)、JCP (Java Community Process)，通过JSR可以就Java某一方面的应用定义一组标准的API或者服务。对于最终用户来说，他们的代码只需要调用JSR定义的标准API，不做任何修改就可以调用不同的JSR实现。这里常见的例子就是Java Servlet应用，用户开发的Web应用可以不做任何修改就部署到Tomcat、JBoss等不同的Web容器中。

JAXRS是JCP为Java RESTful Web Service定义的一套API。由于Web服务的描述模型与Java类和接口有一定的差距，JAX-RS定义了很多annotation，通过这些annotation我们可以很方便地将Java类描述成为相关的REST服务。由于RESTful Web Service通常需要部署到Web容器中，JAX-RS也定义了相关服务来发现部署到容器中的JAX-RS应用。

读过JSR规范的朋友或多或少都会有这样的体会，JSR作为规范文档，其目标是将API定义以及实现功能描述清楚、完备，其目标读者是相关API的实现人员，或者是相关API的高级使用人员。如果读者对相关的背景知识还不熟悉的话，JSR文档读起来会比较晦涩而且难以理解。加之绝大部分JSR文档都没有相关的中文翻译，对于绝大多数初学者来说，通过阅读JSR文档来学习相关的API的知识是一个艰难的过程。

如果我们想要对JAX-RS规范有一个比较快速并且全面的了解应该怎么办呢？一般来我们可以通过JSR的相关参考实现入手，我们不但可以通过运行相关的参考实现的例子快速入门，还可以通过跟踪相关的代码对实现细节有一个全面的了解。韩陆的这本新作以JAX-

RS 的参考实现 Jersey 为蓝本, 由浅入深地向大家介绍了 JAX-RS 的由来, 以及与 RESTful Web 服务开发的相关 API, 并结合实例分享了作者的实战经验。

好了, 现在打开你熟悉的 IDE 工具, 加载 Jersey 代码库, 沿着本书的指引去探索 Java RESTful Web Services 开发世界吧。

RedHat 姜宁

Preface 前言

本书第1版发行后, Jersey 版本从 2.9 更新到了 2.22.2, 此间 REST 服务得到了更广泛的认可和使用。与此同时, Java 8、Spring Boot 和 Docker 的爆发式发展, 使得 Java 领域的 RESTful 开发有了新的发展。

第2版变更

迫不及待, 这是我想为读者更新 REST 服务新发展的决心, 遂有此第2版。首先, 我们要拥抱 Java 8。lambda 表达式在大数据处理, 尤其在 Spark 中是默认的语法表达; Java 8 带给我们的不只是“语法糖”, 而是开发和执行效率的提升。我从实践中得到了其中的好处, 也希望读者能跟上时代的步伐。其次是 Spring Boot, 这是 Java 领域实现微服务的事实标准框架。我已经无法回去适应部署 war 到 Tomcat 的时代, 请保守的读者原谅我的情不自禁。再次是 Docker, 我希望读者具备使用 Docker 完成开发自测阶段的一切, 也希望读者能运用 Docker 实现微服务的部署和可伸缩实践。

从第1版第1次印刷至今, 我始终关注着读者的反馈。邮件都做了认真的回复。根据读者的反馈, 我在第2版中重新梳理了章节的结构, 删除了第1版中反馈不好的第9章和第11章, 调整后的章节与第1版的对应关系如下。

- 第1章合并了第1版的第1章和第2章。
- 第2章对应第1版第3章。
- 第3章对应第1版第4章。
- 第4章包含了第1版的第8章。
- 第5章在第1版的基础上做了更新。
- 第6章包含了第1版的第7章, 并升级了第1版 2.5 节的示例。
- 第7章和第8章是新增章节。

□第9章对应第1版第10章。

□第10章包含了第1版的第6章。

与许多技术作者一样，写书的时间是挤出来的。如果精力尚可，每晚7点到9点、11点到凌晨2点是我动笔的时间，偶尔，早上6点到8点我也会赶赶。写书成为我梳理、总结和思考的最佳方式。

于此过程，我总结了3句话与读者共享。搞技术的人，是停不下来的。时而要开疆拓土，学习和研究新的知识点，弥补自己的技术债；时而要运筹帷幄，将知识点梳理成线，编织成网；时而要深耕细作，面对当下要攻坚的业务所对应的知识点，深入研究、反复实践、勤于思考、勇于交流。只有这样，我们才可以坦然地用手推一下眼镜，谦虚地告诉别人，“其实我是个程序员”。

源代码

本书提供源代码下载，地址是 <https://github.com/feuyeux/jax-rs2-guide-II>。

勘误和交流

本书的勘误会在 <https://github.com/feuyeux/jax-rs2-guide-II/wiki> 发布，欢迎读者批评指正。

□我的邮箱：feuyeux@163.com

□我的新浪微博：六爷 1_1

致谢

感谢我的妻子 Caroline 和女儿 Doris 一直以来的关心和陪伴。

感谢华章公司的杨福川对我的专业指导。感谢华章公司编辑高婧雅、李艺专业和耐心的审阅和指正。

感谢阿里巴巴速卖通中间件团队在微服务、容器化上对我的影响。感谢雷卷、许晓斌在DDD、Spring Boot 和 Docker 上对我的帮助。感谢 Technicolor 的敏捷团队、阿里巴巴国际站测试架构团队，前者带我悟得 Jersey，后者给我深入实践的机会。

最后我要感谢阿里巴巴阿里云事业群大安全的各位兄弟对我的支持。我正在这里，与大家一天天、一步步将微服务和容器化落地生花。

Contents 目 录

第 2 版序一

第 2 版序二

第 1 版序一

第 1 版序二

前言

第 1 章 JAX-RS2 入门..... 1

1.1 解读 REST 1

1.1.1 一种架构风格..... 2

1.1.2 基本实现形式..... 2

1.2 解读 REST 服务..... 3

1.2.1 REST 式的 Web 服务..... 3

1.2.2 对比 RPC 风格..... 3

1.2.3 对比 MVC 风格..... 4

1.3 解读 JAX-RS 标准..... 5

1.3.1 JAX-RS2 标准..... 5

1.3.2 JAX-RS2 的目标..... 5

1.3.3 非 JAX-RS2 的目标..... 6

1.3.4 解读 JAX-RS 元素..... 7

1.4 Jersey 项目概要..... 7

1.4.1 获得 Jersey..... 8

1.4.2 Jersey 问答..... 8

1.4.3 Jersey 项目管理..... 8

1.4.4 Jersey 许可..... 9

1.4.5 Jersey 的模块..... 10

1.4.6 GlashFish 项目..... 10

1.5 快速实现 Java REST 服务..... 12

1.5.1 第一个 REST 服务..... 13

1.5.2 第一个 Servlet 容器服务..... 17

1.6 快速了解 Java REST 服务..... 19

1.6.1 REST 工程类型..... 19

1.6.2 REST 应用描述..... 24

1.7 Java 领域的其他 REST 实现..... 27

1.7.1 JAX-RS 的其他实现..... 27

1.7.2 其他的 REST 实现..... 31

1.8 REST 调试工具..... 33

1.8.1 命令行调试工具..... 33

1.8.2 基于浏览器的图形化
调试插件..... 34

1.9 本章小结..... 37

第 2 章 REST API 设计..... 38

2.1 统一接口..... 38

2.1.1 GET 方法..... 39

2.1.2 PUT 方法..... 41

2.1.3 DELETE 方法..... 43

2.1.4 POST 方法..... 44

2.1.5 WebDAV 扩展方法..... 45

2.2 资源定位..... 47

2.2.1 资源地址设计..... 48

2.2.2 @QueryParam 注解..... 50

2.2.3 @PathParam 注解..... 52

| | | | | | |
|------------------------|-------------------------|-----|-------------------------|----------------------|-----|
| 2.2.4 | @FormParam 注解 | 55 | 3.4.4 | ClientResponseFilter | 105 |
| 2.2.5 | @BeanParam 注解 | 57 | 3.4.5 | 访问日志 | 107 |
| 2.2.6 | @CookieParam 注解 | 58 | 3.5 | REST 拦截器 | 109 |
| 2.2.7 | @Context 注解 | 58 | 3.6 | 绑定机制 | 111 |
| 2.3 | 传输格式 | 59 | 3.6.1 | 名称绑定 | 111 |
| 2.3.1 | 基本类型 | 59 | 3.6.2 | 动态绑定 | 113 |
| 2.3.2 | 文件类型 | 60 | 3.7 | 优先级 | 115 |
| 2.3.3 | InputStream 类型 | 61 | 3.8 | 本章小结 | 116 |
| 2.3.4 | Reader 类型 | 62 | | | |
| 2.3.5 | XML 类型 | 62 | 第 4 章 REST 服务与异步 | 117 | |
| 2.3.6 | JSON 类型 | 66 | 4.1 | 为什么使用异步机制 | 117 |
| 2.4 | 连通性 | 82 | 4.1.1 | 服务器异步机制 | 117 |
| 2.4.1 | 过渡型链接 | 82 | 4.1.2 | 客户端异步机制 | 118 |
| 2.4.2 | 结构型链接 | 83 | 4.2 | JAX-RS2 的异步机制 | 119 |
| 2.5 | 处理响应 | 84 | 4.2.1 | 服务端实现 | 119 |
| 2.5.1 | 返回类型 | 85 | 4.2.2 | 客户端实现和测试 | 122 |
| 2.5.2 | 处理异常 | 86 | 4.3 | 基于 HTTP1.1 的异步通信 | 124 |
| 2.6 | 内容协商 | 89 | 4.3.1 | Polling 技术 | 124 |
| 2.6.1 | @Produces 注解 | 89 | 4.3.2 | Comet 技术 | 126 |
| 2.6.2 | @Consumes 注解 | 91 | 4.3.3 | Web Hook 异步通信 | 127 |
| 2.7 | 本章小结 | 92 | 4.3.4 | SSE 技术 | 128 |
| 第 3 章 REST 请求处理 | 93 | | 4.4 | 基于 HTML5 的异步通信 | 129 |
| 3.1 | Jersey 的 AOP 机制 | 93 | 4.4.1 | SSE 的原理 | 129 |
| 3.2 | Providers 详解 | 94 | 4.4.2 | 发布—订阅模式的实现 | 131 |
| 3.2.1 | 实体 Providers | 94 | 4.4.3 | 广播模式的实现 | 135 |
| 3.2.2 | 上下文 Providers | 100 | 4.4.4 | WebSocket 技术 | 137 |
| 3.3 | REST 请求流程 | 100 | 4.5 | 本章小节 | 138 |
| 3.4 | REST 过滤器 | 102 | 第 5 章 REST 客户端 | 139 | |
| 3.4.1 | ClientRequestFilter | 102 | 5.1 | 客户端接口 | 140 |
| 3.4.2 | ContainerRequestFilter | 103 | 5.1.1 | Client 接口 | 140 |
| 3.4.3 | ContainerResponseFilter | 104 | 5.1.2 | WebTarget 接口 | 141 |

| | | |
|-------|--------------------|-----|
| 5.1.3 | Invocation 接口 | 142 |
| 5.2 | 连接池 | 142 |
| 5.2.1 | 资源释放 | 142 |
| 5.2.2 | 连接器 | 144 |
| 5.2.3 | HTTP 连接池 | 146 |
| 5.3 | 封装 Client | 147 |
| 5.4 | 请求 Spring Boot 微服务 | 148 |
| 5.4.1 | 不同的 JSON 解析方式 | 148 |
| 5.4.2 | 完整示例 | 150 |
| 5.5 | JavaScript 客户端 | 150 |
| 5.5.1 | jQuery 客户端 | 151 |
| 5.5.2 | AngularJs 客户端 | 152 |
| 5.6 | 本章小结 | 152 |

第 6 章 REST 测试 153

| | | |
|-------|-----------------|-----|
| 6.1 | Jersey 测试框架 | 153 |
| 6.2 | 单元测试 | 156 |
| 6.2.1 | 集成 Spring 的单元测试 | 156 |
| 6.2.2 | 异步测试 | 158 |
| 6.3 | 集成测试 | 158 |
| 6.4 | 日志增强 | 159 |
| 6.5 | 本章小结 | 160 |

第 7 章 微服务 161

| | | |
|-------|----------------------|-----|
| 7.1 | 微服务技术栈 | 162 |
| 7.1.1 | 服务发现 | 163 |
| 7.1.2 | 可伸缩性 | 163 |
| 7.1.3 | 回到起点 | 164 |
| 7.2 | REST 服务与 Spring Boot | 165 |
| 7.2.1 | Bootiful | 165 |
| 7.2.2 | RESTful | 167 |
| 7.2.3 | Actuator | 168 |

| | | |
|-------|------------------------|-----|
| 7.3 | REST 服务与 Spring Cloud | 172 |
| 7.3.1 | Spring Cloud Zookeeper | 172 |
| 7.3.2 | Spring Cloud Consul | 182 |
| 7.3.3 | Spring Cloud Etcd | 187 |
| 7.4 | 本章小结 | 193 |

第 8 章 容器化 195

| | | |
|-------|------------|-----|
| 8.1 | 容器技术 | 195 |
| 8.1.1 | 容器 | 195 |
| 8.1.2 | Docker 技术栈 | 197 |
| 8.1.3 | 容器文化 | 199 |
| 8.2 | REST 服务与容器 | 201 |
| 8.2.1 | 开始容器化之路 | 201 |
| 8.2.2 | 开发自测容器化 | 204 |
| 8.3 | 容器化微服务 | 206 |
| 8.3.1 | Zookeeper | 207 |
| 8.3.2 | Kafka | 212 |
| 8.3.3 | 微服务 | 214 |
| 8.3.4 | Nginx | 217 |
| 8.4 | 本章小结 | 220 |

第 9 章 JAX-RS 调优 223

| | | |
|-------|------------|-----|
| 9.1 | 使用缓存优化负载 | 223 |
| 9.1.1 | 缓存协商 | 223 |
| 9.1.2 | 条件 GET | 225 |
| 9.1.3 | REST 缓存实践 | 227 |
| 9.1.4 | ab 测试 | 229 |
| 9.2 | 使用版本号优化服务 | 229 |
| 9.2.1 | 何时使用版本号 | 230 |
| 9.2.2 | 如何使用版本号 | 230 |
| 9.3 | 使用参数配置优化服务 | 232 |
| 9.3.1 | 通用配置 | 232 |

| | |
|-------------------|-----|
| 9.3.2 服务器端和客户端配置类 | 233 |
| 9.4 Java 虚拟机调优 | 234 |
| 9.4.1 虚拟机概述 | 234 |
| 9.4.2 内存溢出与内存泄漏 | 236 |
| 9.5 本章小结 | 238 |

第 10 章 REST 安全 239

| | |
|------------------------|-----|
| 10.1 身份认证 | 240 |
| 10.1.1 基本认证 | 241 |
| 10.1.2 摘要认证 | 241 |
| 10.1.3 表单认证 | 242 |
| 10.1.4 证书认证 | 242 |
| 10.2 资源授权 | 244 |
| 10.2.1 容器管理权限 | 244 |
| 10.2.2 应用管理权限 | 246 |
| 10.3 认证与授权实现 | 247 |
| 10.3.1 基本认证与 JDBCRealm | 247 |

| | |
|---------------------------------|-----|
| 10.3.2 摘要认证与 UserDatabase-Realm | 255 |
| 10.3.3 表单认证与 DataSource-Realm | 258 |
| 10.3.4 Form 认证和 JAASRealm | 263 |
| 10.3.5 证书认证与 UserDatabase-Realm | 266 |
| 10.4 JAX-RS2 实现 | 270 |
| 10.4.1 Application 类 | 270 |
| 10.4.2 资源类 | 271 |
| 10.4.3 资源测试类 | 271 |
| 10.5 REST 服务与 OAuth2 | 273 |
| 10.5.1 OAuth2 概述 | 274 |
| 10.5.2 OAuth2 流程 | 275 |
| 10.5.3 OAuth2 实现 | 276 |
| 10.6 本章小结 | 280 |

参考资料 282

1.2 解读 REST 服务



JAX-RS2 入门

本章将详细讲述 REST 服务 (RESTful Web Service) 的概念、生态环境, 并通过简单的示例, 使读者快速掌握 REST 服务开发的基本能力。

前四节将逐一解读 REST 的概念、REST 服务、JAX-RS 标准和 Jersey 项目。这四者之间的联系是: REST 是一种跨平台、跨语言的架构风格, REST 式的 Web 服务是对 REST 在 Web 领域的实现; JAX-RS 标准是 Java 领域对 REST 式的 Web 服务制定的实现标准, Jersey 是 JAX-RS 标准的参考实现, 是 Java EE 参考实现项目 GlassFish 的成员项目。接下来的三节介绍基于 Jersey 的 REST 服务开发, 以及 Java 领域中其他的 REST 服务框架。最后, 介绍 REST 服务的调试工具。

1.1 解读 REST

REST (Representational State Transfer) 翻译为表述性状态转移, 源自 Roy Thomas Fielding 博士在 2000 年就读加州大学欧文分校期间发表的一篇学术论文《Architectural Styles and the Design of Network-based Software Architectures》。REST 之父在该论文中提出了 REST 的 6 个特点, 分别是: 客户端-服务器的、无状态的、可缓存的、统一接口、分层系统和按需编码。

REST 具有跨平台、跨语言的优势。从其诞生开始, 就得到了诸多语言的快速支持, 最著名的是 ROR (Ruby on Rails) 框架。新兴的语言 (比如 NodeJs、Golang)、工具平台 (Docker、Spark) 和公有云, 更是将 REST 默认为服务的开放形式。

1.1.1 一种架构风格

REST 是一种架构风格。在这种架构风格中，对象被视为一种资源（resource），通常使用概念清晰的名词命名。

表述性状态是指资源数据在某个瞬时的状态快照。资源可以有多种表述（representation），表述状态具有描述性，包括资源数据的内容、表述格式（比如 XML、JSON、Atom）等信息。

REST 的资源是可寻址的，通过 HTTP1.1 协议（RFC 2616）定义的通用动词方法（比如 GET、PUT、DELETE、POST），使用 URI 协议（RFC3305）来唯一标识某个资源公布出来的接口。

请求一个资源的过程可以理解为访问一个具有指定性和描述性的 URI，通过 HTTP 协议，将资源的表述从服务器“转移”到客户端或者相反方向。

阅读指南

REST 不是一种技术（technology），也不是一个标准（standard）/ 协议（protocol），而是一种使用既有标准：HTTP+URI+XML（XML 似乎成为了数据格式的借指，不仅代表 XML 本身）来实现其要求的架构风格。因此，与之对应的不是 SOAP 协议，而是像 RPC 这样的架构风格。

1.1.2 基本实现形式

HTTP+URI+XML 是 REST 的基本实现形式，但不是唯一的实现形式。REST 一开始便使用已有的 HTTP 协议（RFC 2616）、URI 协议（RFC3305）来描述其特征，而对如何使用一种编程语言来实现，并没有进行任何描述和规定，甚至应该包含哪些传输类型或者数据格式也没有描述，但通常的实现至少包含 XML 格式。

具体而言，HTTP 协议和 URI 用于统一接口和定位资源，文本、二进制流、XML 和 JSON 等格式用来作为资源的表述。正如采用已有技术 XMLHttpRequest+JavaScript+XML（XML 后来几乎被 JSON 替代）实现 Ajax 一样，使用 HTTP+URI+XML 实现 REST 的好处是让开发者持有这些已知的技术来开发 REST 的入门门槛较低，关注点更容易放到 REST 的核心概念和业务逻辑上。

阅读指南

以 HTTP+URI+XML 实现的应用并不一定是 REST 服务，但对于 Ajax，这个逆命题是成立的。因为 Ajax 是一种技术，而 REST 是一种架构风格。学习和使用 REST 的关键是掌握这种思想，而不是具体的实现形式。

1.2 解读 REST 服务

RESTful 对应的中文是 REST 式的，RESTful Web Service 的准确翻译应该是 REST 式的 Web 服务，我们通常简称为 REST 服务。RESTful 的应用或者 Web 服务是最常见的两种 REST 式的项目部署、存在的方式。本节将介绍 REST 服务并对比其与传统 Web Services 的不同。

1.2.1 REST 式的 Web 服务

RESTful Web Service 是一种遵守 REST 式风格的 Web 服务。REST 服务是一种 ROA (Resource-Oriented Architecture, 面向资源的架构) 应用。其主要特点是方法信息存在于 HTTP 协议的方法中 (比如 GET、PUT)，作用域存在于 URI 中。例如，在一个获取设备资源列表的 GET 请求中，方法信息是 GET，作用域信息是 URI 中包含的对设备资源的过滤、分页和排序等条件。

1.2.2 对比 RPC 风格

相比 Web 服务领域广为流行的 RPC (Remote Procedure Call Protocol, 远程过程调用协议) 风格，REST 风格更轻量和快速。从方法信息角度看，REST 采用标准的 HTTP 方法，而 RPC 请求都是 HTTP 协议的 POST 方法，其方法信息包含于 SOAP 协议包或 HTTP 协议包中，方法名称不具有通用性。从作用域角度看，REST 采用 URI 显式定义作用域，而 RPC 的这一信息同样包含于协议包中，不能直观呈现。

RPC 风格的开发关注于服务器 - 客户端之间的方法调用，而不关注基于哪个网络层的哪种协议。也就是说，RPC 是面向方法调用过程的，相比而言，REST 是面向资源状态的。RPC 风格的两个代表是 XML-RPC 和大 Web 服务。

1. XML-RPC

XML-RPC 是一种使用 XML 格式封装方法调用，并使用 HTTP 协议作为传送机制的 RPC 风格的实现。XML-RPC 的请求方法都是 HTTP 协议的 POST 方法，请求和响应的数据格式均为 XML。

XML-RPC 的数据格式和使用 XML 作为资源的表述的 REST 外观上很相似，但数据的内容则大相径庭。REST 式的 XML 信息的主体是对一个资源状态的表述，无须包含方法信息，因为其请求的 HTTP 方法就已经决定了这一点。XML-RPC 的请求数据结构额外包含方法调用信息和参数信息。

对于响应信息的内容两者也截然不同，REST 式通常会包含响应实体信息，以及 HTTP

状态码和可选的异常信息，而 XML-RPC 的返回信息仅仅是对方法调用的响应信息。

XML-RPC 是一种遗留技术，已经被 SOAP 取代。在 Java 领域，JAX-RPC 标准已经并入 JAX-WS2 标准。XML-RPC 的应用依然存在，著名的测试用例管理系统 TestLink 的对外接口就是使用 PHP 开发的 XML-RPC。

2. 大 Web 服务

大 Web 服务 (Big Web Service) 是 Leonard Richardson 和 Sam Ruby 在其所著的《RESTful Web Services》一书中，对基于 SOAP+ WSDL+UDDI+WS- 标准栈等技术实现 RPC 风格的大型 Web 服务的统称。事实上，“大 Web 服务”这一说法也被 Java EE 7 的布道者们在多次演讲中使用。在 Java 领域，对应的标准主要是 JAX-WS 2.0/2.1/2.2 (JSR 224)。相较 REST 式的 Web 服务，大 Web 服务功能更强大，设计更复杂。大 Web 服务同样是跨平台、跨语言的，对复杂的数据类型的支持也非常好。大 Web 服务是基于 RPC 风格的重量设计，因此方法和作用域无法通过直观断定，需要定义在消息中，而且方法名不是统一和通用的。同时，大 Web 服务走 HTTP 协议时，请求都是基于 POST 方法的。

对比 RPC 风格的 Web 服务，REST 式的 Web 服务形式更简单、设计更轻量、实现更快捷。REST 无须引入 SOAP 消息传输层，无须注册服务，也没有客户端 stub 的概念等。但是，REST 式的 Web 服务并没有像大 Web 服务那样提供诸如安全策略等全面的标准规范。

大 Web 服务和 REST 式的 Web 服务各有优势，并不是一种替换关系。在实际开发中，两者共存于一个项目中也是一种解决方案。

1.2.3 对比 MVC 风格

MVC 风格的出现将模型、视图、控制解耦，其亮点是从前到后的一致性，其结构整洁、逻辑清晰，易于扩展和增强。MVC 在 Java 领域的普遍实现方式是在 Web 前端使用标签库来对应服务端的模型类实例和控制类实例，标签库和服务端依赖库可以是松散的耦合，比如 Spring 生态系统，也可以是全栈式的统一体系，比如 JSF 体系。但无论如何实现，在 Web 前端的开发过程中，必须时刻考虑页面标签和服务端的映射关系，包括模型类的匹配和转换、数据结构、控制类的输入和输出的参数类型和数量等。

因此，MVC 风格偏重于解决服务器端的逻辑分层问题，以及客户端是逻辑分层的延伸问题。MVC 的标签库虽然其形态已经和 HTML 页面融合，但本质上还是 Java 编写的装饰模式的类实例，对应的是服务器端使用 Java 编写的模型类或者控制器类，因此 MVC 很难实现跨语言解耦。而 REST 风格偏重于统一接口，因此具体实现就可以跨平台和跨语言。REST 推动了 Web 开发的新时代，使用平庸的纯 HTML 作为客户端，没有服务器端和客户端的耦合。显而易见，使用纯 HTML 开发的 REST 客户端和使用 Java 开发的 REST 服务器

端并不存在语言上的耦合。

MVC 和 REST 式并不是互斥的, 如 Spring 的 MVC 模块已经开始支持 REST 式的开发。Jersey 作为 JAX-RS 标准的实现, 也实现了 MVC 的功能, 请参考相关模块: jersey-mvc、jersey-mvc-freemarker 和 jersey-mvc-jsp。需要说明的是, 本书致力于讲述 JAX-RS, 对于 Jersey 实现中的 JAX-RS 之外的功能, 只做必要的讲述。由于 MVC 和 REST 之间有更多的并行存在性, 本书余文没有将 MVC 放入讲述之列。

1.3 解读 JAX-RS 标准

JAX-RS 是 Java 领域的 REST 式的 Web 服务的标准规范, 是使用 Java 完成 REST 服务的基本约定。

1.3.1 JAX-RS2 标准

Java 领域中的 Web Service 是指实现 SOAP 协议的 JAX-WS。直到 Java EE 6 (发布于 2008 年 9 月) 通过 JCP (Java Community Process) 组织定义的 JSR311 (<http://www.jcp.org/en/jsr/detail?id=311>), 才将 REST 在 Java 领域标准化。

JSR311 名为 The Java API for RESTful Web Service, 即 JAX-RS, 其参考实现是 Glassfish 项目中的 Jersey1.0。此后, JSR311 进行了一次升级 (2009 年 9 月), 即 JAX-RS1.1。JAX-RS 诞生后, 时隔 5 年 (2013 年 5 月) 发布的 Java EE7 包含了 JSR339, 将 JAX-RS 升级到 JAX-RS2 (<http://www.jcp.org/en/jsr/detail?id=339>)。JAX-RS2.0 在前面版本的基础上增加了很多实用性的功能, 比如对 REST 客户端 API 的定义, 异步 REST 等, 对 REST 的支持更加完善和强大。

JAX-RS 的版本对应的参考实现 Jersey 项目版本信息参见表 1-1。

表 1-1 JAX-RS 标准和 Jersey 版本信息

| JSR 标准 | JSR 名称 | 标准发布时间 | JSR 实现 |
|--------|------------|-----------------|-----------|
| jsr311 | JAX-RS 1.0 | 2008 年 9 月 8 日 | Jersey1.x |
| jsr311 | JAX-RS 1.1 | 2009 年 9 月 17 日 | Jersey1.x |
| jsr339 | JAX-RS 2.0 | 2013 年 5 月 22 日 | Jersey2.x |

1.3.2 JAX-RS2 的目标

JAX-RS2 标准 (即 JSR339) 中定义了目标、非目标和元素等内容。JSR339 标准中的这部分内容通常被以实现业务功能为目的的开发人员所忽视, 在此和读者分享的一个开发经验

是：要掌握一项技术，先要掌握它背后标准的定义。首先我们来看看 JAX-RS2 的目标。

1) 基于 POJO：JAX-RS2 的 API 提供一组注解 (annotation) 和相关的接口、类，并定义了 POJO (Plain Ordinary Java Object) 对象的生命周期和作用域。规定使用 POJO 来公布 Web 资源。

2) 以 HTTP 为中心：JAX-RS2 采用 HTTP 协议，并提供清晰的 HTTP 和统一资源定位 (URI) 元素来映射相关的 API 类和注解。JAX-RS2 的 API 不但支持通用的 HTTP 使用模式，还对 WebDAV 和 Atom 等扩展协议提供灵活的支持。

3) 格式独立性：JAX-RS2 对传输数据 (HTTP Entity) 的类型 / 格式的支持非常宽泛，允许在标准风格之上使用额外的数据类型。

4) 容器独立性：JAX-RS2 的应用可以部署在各种 Servlet 容器中，比如 Tomcat/Jetty，也可以部署在支持 JAX-WS 的容器中，比如 GlassFish。

5) 内置于 Java EE：JAX-RS2 是 Java EE 规范的一部分，它定义了在一个 Java EE 容器内的 Web 资源类的内部，如何使用 Java EE 的功能和组件。

阅读指南

WebDAV (Web-based Distributed Authoring and Versioning，基于 Web 的分布式创作和版本控制) 是 IETF 组织的 RFC2518 协议。WebDAV 基于并扩展了 HTTP 1.1，在 HTTP 标准方法以外添加了以下内容。

❑ Mkcol：创建集合。

❑ PropFind/PropPatch：针对资源和集合检索和设置属性。

❑ Copy/Move：管理命名空间上下文中的集合和资源。

❑ Lock/Unlock：改写保护，支持文件的版本控制。

针对在 REST 风格的 Web 服务中是否应该使用 WebDAV，业内的声音并不一致，持反对意见的主要观点是 WebDAV 带来了非统一的接口，这违背了 REST 的初衷。本书的示例将不采用 WebDAV，但文字部分将讲述如何支持 WebDAV。Atom 类型传输格式将在 2.3 节讲述。

1.3.3 非 JAX-RS2 的目标

那么哪些不是 JAX-RS2 的目标呢？

1) 对 J2SE 6.0 之前版本的支持：JAX-RS2 中大量使用了注解 (annotation)，需要 J2SE 6.0 以及更新的版本，因此不提供对 J2SE 6.0 以下版本的支持。

2) 对服务的描述、注册和探测：JAX-RS2 没有定义也无须支持任何服务的描述

(description)、服务的注册 (registration) 和服务的探测 (discovery)。

3) HTTP 协议栈: JAX-RS2 没有定义新的 HTTP 协议栈。承载 JAX-RS2 应用的容器提供对 HTTP 协议的支持。

4) 数据类型 / 格式类: JAX-RS2 没有定义处理实体内容的类, 它将这一类型的类交由使用 JAX-RS2 的应用中的类去实现。

1.3.4 解读 JAX-RS 元素

最后, 我们来看看 JAX-RS2 中定义了哪些元素。

1) 资源类: 使用 JAX-RS 注解来实现相关 Web 资源的 Java 类。如果用 MVC 的三层结构来解读, 那么资源类位于最前端, 用于接收请求和返回响应。通常, 但不是约定, 我们使用 resource 作为包名, 三层的包定义形如: resource-service-dao。

2) 根资源类: 使用 @Path 注解, 提供资源类树的根资源及其子资源的访问。资源类分为根资源类和子资源类, 由于 Jersey 默认提供 WADL (参见 1.6 节), 每个应用公布的全部资源接口可以通过 WADL 页面查阅。

3) 请求方法标识符: 使用运行期注解 @HttpMethod, 用来标识处理资源的 HTTP 请求方法。该方法将使用资源类的相应方法处理, 标准的方法包括 DELETE、GET、HEAD、OPTIONS、POST、PUT, 详见 2.1 节。

4) 资源方法: 资源类中定义的方法使用了请求方法标识符, 用来处理相关资源的请求。就是上面提到的资源类的相应方法。

5) 子资源标识符: 资源类中定义的方法, 用来定位相关资源的子资源。

6) 子资源方法: 资源类中定义的方法, 用来处理相关资源的子资源的请求。

7) Provider: 一种 JAX-RS 扩展接口的实现类, 扩展了 JAX-RS 运行期的能力。第 4 章详述了各种 Provider 及其实现。

8) Filter: 一种用于过滤请求和响应的 Provider, 详见 3.3 节。

9) Entity Interceptor: 一种用于处理拦截消息读写的 Provider, 详见 3.5 节。

10) Invocation: 一种用于配置发布 HTTP 请求的客户端 API 对象, 详见 5.1.3 节。

11) WebTarget: 一种使用 URI 标识的 Invocation 容器对象, 详见 5.1.2 节。

12) Link: 一种携带元数据的 URI, 包括媒体类型、关系和标题等, 详见 2.4 节。

1.4 Jersey 项目概要

Jersey 是 JAX-RS 标准的参考实现, 是 Java 领域中最纯正的 REST 服务开发框架。本

节将带读者走近 Jersey 的世界。

Jersey 项目是 GlashFish 项目的一个子项目，专门用来实现 JAX-RS (JSR 311 & JSR 339) 标准，并提供了扩展特性。

1.4.1 获得 Jersey

Jersey 项目的地址是 <https://jersey.java.net>。该网站同时提供了 JAX-RS 和 JAX-RS2 两个并行版本，分别是 JAX-RS1.1（截至本书发稿，最新版本是 Jersey1.19）和 JAX-RS2（截至本书发稿，最新版本是 Jersey2.22.1）。读者可以通过单击 latest Jersey User Guide 获取和阅读最新版本的用户手册，这是官方发布的第一手参考资料。

Jersey 项目的下载地址 <http://jersey.java.net/download.html>。该页面自上而下的内容分别如下。

- JAX-RS 标准列表链接 (JAX-RS 2.0 API)。
- Jersey 最新参考实现的 jar 包下载 (Jersey JAX-RS 2.0 RI bundle)。
- Jersey 最新参考实现的示例代码下载 (Jersey 2.22.1 Examples bundle)。
- 通过 Maven 模板 (archetype)，使用 Jersey 最新版本创建 REST 服务的命令。
- Jersey 最新参考实现的模块和依赖 (Jersey 2 modules and dependencies)。
- JAX-RS1.1 的参考实现包下载。

Jersey 源代码的托管地址是 <https://github.com/jersey/jersey>，我们可以通过 git 命令，将 Jersey 主干代码迁出到本地。示例如下。

```
git clone https://github.com/jersey/jersey.git
```

1.4.2 Jersey 问答

StackOverflow 是专业的程序员问答系统，Jersey 的问题列表地址是：<http://stackoverflow.com/questions/tagged/jersey>。该链接在 Jersey 官网首页底部被列出，可见 Jersey 对问答系统的重视。另外，邮件列表也是一种知识共享的途径，读者可以自行订阅，地址是：<https://jersey.java.net/mailling.html>。

1.4.3 Jersey 项目管理

Jersey 使用 JIRA 作为项目管理平台，相应的地址是：<https://java.net/jira/browse/JERSEY>。JIRA 和 StackOverflow 不同的是，JIRA 平台是 Jersey 团队日常开发的管理平台，即 Jersey 官方的缺陷管理平台，用于上报缺陷和改进意见，而不是社区性质的交流平台。通过这个平台我们可以从中了解到 Jersey 项目的进展情况。Jersey 是一个非常活跃的项目，

不仅可以从 github 源代码的提交活动中看到该项目频繁的更新，在 JIRA 中也可以看到该项目推进的速度。

这里为喜欢开源社区活动的读者举个例子。在撰写本书第 1 版的开始，Jersey2.0 并不支持与 Spring 的集成，因为 Jersey 的 IoC 容器由 GlashFish 的另一个子项目 HK2 来支持。随后，我在 JIRA 上发现一个 Jersey2.x 支持与 Spring 集成的任务被创建了（<https://java.net/jira/browse/JERSEY-1957>），此后我经常观察其进展状态，最终看到了这个功能在 Jersey2.2 中以扩展包的形式发布了。

因此，在使用 Jersey 的过程中，如果读者遇到 Jersey 本身的问题，可以跟踪 Jersey 的 JIRA 平台检索、查看 Bug 的修复状态，包括将在哪个版本修复，有什么样的临时解决办法（workaround）。同时，跟踪 JIRA 也可以了解新版本的发布情况，包括新增哪些功能，升级对哪一部分带来性能、安全的提升等。换句话说，JIRA 展示了 Jersey 项目的缺陷修复和新功能发版的计划（roadmap）。

1.4.4 Jersey 许可

开发者使用开源软件的前提是了解它的许可证版本，否则可能会带来侵权问题。相信在正规的公司，大家都有被开发管理部门的人“恐吓”的经历。开发者需感谢这样的团队所做的工作，他们为公司规避了商业侵权的风险，因为引用的源代码如果出自“传染性”许可，该项目是不能用于闭源的商业用途的。

Jersey 的许可证说明地址是：<https://jersey.java.net/license.html>。从中我们可以了解到 Jersey 使用的是双许可证：CDDL（Common Development and Distribution License，开源通用开发和分发许可证）1.1 和 GPLv2（类路径例外）许可证。双重许可是依照两套（或更多套）不同的条款和条件分发相同软件的做法。在为软件授予双重许可时，接收人可以选择他们希望依照哪种条款获得软件。使用双重许可的两个常见动机是遵循商业模式和保持许可证兼容性。GPLv2.0 许可证为无法依照 CDDL 许可证使用 Jersey 的供应商提供了一个额外选项。Jersey 许可证使整套产品和包保持一致（GlassFish 项目同样依照 CDDL 和 GPLv2（类路径例外）授予双重许可）。

阅读指南

类路径例外是由自由软件基金会的 GNU/ 类路径项目制订的。它允许将依照任何许可证提供的应用程序链接到依照 GPLv2 许可的软件中包含的库，而该应用程序不受 GPL 要求公开其本身的限制。

为什么需要使用类路径例外？因为作为“基于 GPL 程序的作品”的一部分提供的所有

代码还应获得 GPL 许可。因此，需要指定 GPL 许可证例外的情况，以便明确将链接到 GPL 实现的任何应用程序从该许可要求中排除。类路径例外就实现了这一目的。

1.4.5 Jersey 的模块

Jersey 框架是由核心模块、容器模块、连接器模块、Media 模块、扩展模块、测试框架模块、安全模块以及 Glassfish Bundle 模块等 8 个大的模块组成。详情请读者浏览官方文档：<https://jersey.java.net/documentation/latest/modules-and-dependencies.html>。

Jersey 核心模块包括 3 个子模块，分别是通用包、服务器端实现包和客户端实现包。Jersey 提供了 3 种 HTTP 容器，分别是 Grizzly2、JDK-HTTP 和 SIMPLE-HTTP，Grizzly2 同时提供了 Servlet 容器。Jersey 客户端底层依赖于连接器来实现网络通信，如果标准的客户端模块功能不能满足业务需求，读者可以考虑引入 Grizzly 连接器包或者 Apache 连接器包。

阅读指南

Jersey 在 2.6 版本做了一次包重构，清除了对 guava 和 ASM 的自然依赖。如果你的项目需要做 Jersey 版本迁移，则需要注意这一点。新的包名为：`jersey.repackaged.com.google.common` 和 `jersey.repackaged.objectweb.asm`。

1.4.6 GlashFish 项目

GlashFish 项目地址为 <https://glassfish.java.net>。GlashFish 著名于世的是 Java EE 服务器项目 Oracle GlassFish Server，该项目还同时包含 Java EE 中的一系列标准规范的参考实现，这些参考实现集成于 GlashFish Server，为其 Java EE 容器提供支持。其中对应 JAX-RS2 的实现项目是 Jersey。

为什么要在 JAX-RS2 的介绍中提及和罗列 GlashFish 项目集呢？因为 Jersey 处于 GlashFish 生态环境中，GlashFish 又是 Java EE 生态环境的参考实现。通过了解 GlashFish 项目，我们可以更好地设计和实现 REST 服务。

这里所列的项目是除 Jersey 以外，其他的 GlashFish 项目，排列顺序并不严谨，大体上以其与 Jersey 的紧密关系降序排列。

❑ HK2 项目：JSR-330 参考实现，项目地址为 <http://hk2.java.net>。HK2 是轻量级 DI 架构，实现 IoC 和 DI 的内核。是 Jersey 实现容器内管理 Bean 的基础。

❑ Grizzly 项目：中文直译为灰熊。JSR-356 参考实现，项目地址为 <https://grizzly.java>。

net。Grizzly 是一个异步 IO 的、高效而健壮的服务器，可以被用作 HTTP 服务器、Servlet 容器，支持 Ajp、Comet、WebSocket 以及相对于 RESTful 的另一种 Web Service 实现 (JAX-WS)。

❑ EclipseLink 项目：该项目实现了多个 JSR 标准，包括 JSR-338/JPA2.1、JSR-222/JAXB2.2、JSR-235/SDO2.1.1、JSR-353/Java API for Processing JSON。项目地址为 <http://www.eclipse.org/eclipselink>。EclipseLink 是 JPA2.1 的一个实现，同时它还实现了其他的 JSR 作为扩展。JPA2.1 是 Java EE 7 的成员，是对 JSR317(JPA2.0) 的升级。JPA2.1 的实现中，最常用的是 JBoss 的 Hibernate，该项目从 4.3 开始实现 JPA2.1。也就是说 Hibernate4.2 是 JPA2.0 的最后一个版本。读者在开发的时候要注意依赖项目版本对标准的支持。JPA 标准还有其他的实现，请参考 http://en.wikipedia.org/wiki/Java_Persistence_API。

❑ Metro 项目：该项目是 JSR 中多个标准的官方实现集，目的是实现全栈式的 Web Service。包括 JSR-224/JAX-WS 2.2、JSR-222/JAXB2.2、JSR-206/JAXP 1.4.6、JSR-067/SAAJ1.3。项目地址为 <https://metro.java.net>。Metro 项目中的多个标准作用各有不同。

- JAX-WS 标准结合了 XML-RPC，使用 SOAP 协议来实现 Web Service。在 JAX-WS 的实现中，不可不提的另外两个实现分别是 Apache 的 CXF 和 Axis。
- WSIT 的前身是 Tango，是一种 JAX-WS 和 .NET 互操作的技术，实现了 WS* 标准。
- SAAJ 规范的作用是基于 SOAP 协议 XML 格式传递带附件的 SOAP 消息。
- JAXP 标准涵盖了 Java 对 XML 过程式处理的诸多技术，包括 DOM、SAX 和 StAX，同时该标准定义了解读 XML 样式的 XSLT。
- JAXB 标准是 Java 处理 XML 和 POJO 映射的技术，是 Jersey 中处理传输数据的重要依赖。

❑ Open MQ 项目：地址 <https://mq.java.net>。Open MQ 是 JMS 2.0 的参考实现。JSR-343 是 Java EE 7 的成员，旨在简化 JMS 的 API。关于消息队列的实现数量，恐怕是其他任何一个标准都望尘莫及的。几乎每一个有能力开发服务器软件、中间件的公司都有自己的 MQ，请参考 http://en.wikipedia.org/wiki/Message_queue。

❑ Mojarra 项目：JSR-344/JSF2 参考实现，项目地址为 <https://javaserverfaces.java.net>。JSF 是一种全栈式的、事件驱动的 B/S 开发模式框架，它包括浏览器端的丰富组件，服务器端覆盖 Java EE 的各种特性。JSF 相对于 Spring，借鉴了其核心思想 IoC 和 AOP，同时给出了标准规范。这有点类似 JPA 借鉴了 hibernate 的 O/R Mapping 思想并标准化。JSF 的另一个实现是 Apache 的 myfaces，当前版本为 2.0.18。另外，

JBoss 的 RichFaces 是基于 JSF 的扩展中最为完善和常用的。更多有关 JSF 的内容和原理,请参考笔者的拙作《JSF2 和 RichFaces4 使用指南》。

❑ OpenJDK 项目:项目地址为 <http://openjdk.java.net>。OpenJDK 是开源的 JDK,从版本 1.7 开始成为官方 JDK 的先行版本,因此是 Java 开发者窥探 Java 发展的第一线的最好资源,同时也是活跃的 Linux 发行版本 Ubuntu 和 Fedora 等默认安装的 JDK 版本。

阅读指南

当前使用的 JDK 版本号的升级规则是从 JDK5.0 发布开始的,Java 升级发布一直采用两种方式发布更新。

❑ 有限升级 (Limited Update) 包含新功能和非安全修正。

❑ 重要补丁升级 (Critical Patch Update, CPU) 只包含安全修正。

有限升级发行序号为 20 的倍数,即一个偶数;重要补丁升级顺延上一个 CPU 的版本号加 5 的倍数并取奇数(必要时加 1)。

举例来说,下一个有限升级的版本号为 7u40,那么接下来的 3 个 CPU 版本号依次为 40+5=7u45, 45+5+1=7u51 和 51+5=7u55。再下一个有限升级的版本号为 7u60,随后的 CPU 版本号依次为 7u65、7u71 和 7u75。

这种命名规则会为重要补丁升级保留几个版本序号,以便新的 CPU 版本号可以取区间值之和而不是在最新版本号上顺延。

1.5 快速实现 Java REST 服务

本节包含两个 Jersey 实战示例,目的是让读者具备快速创建 REST 服务的能力。

在开始实战之前,首先需要读者确认你的环境是否已经安装了 Java 和 Maven。这里使用 Maven 命令,示例如下。

```
mvn -v
```

```
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T19:57:37+08:00)
Maven home: /usr/local/Cellar/maven/3.3.3/libexec
Java version: 1.8.0_40, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.11.1", arch: "x86_64", family: "mac"
```

从 Maven 版本显示命令的结果中,自上而下可以看到 Maven 的版本信息和 HOME 路

径信息、Java 的版本信息和 HOME 路径信息、本地语言、平台字符集以及操作系统信息。

1.5.1 第一个 REST 服务

Jersey 提供了 Maven 原型 (archetype) 来快速创建 REST 服务项目。

1. 创建项目

我们首先使用 archetypeGroupId 为 org.glassfish.jersey.archetypes 的原型、artifactId 为 jersey-quickstart-grizzly2 的原型，创建 REST 服务项目。示例如下。

```
mvn archetype:generate \
-DarchetypeArtifactId=jersey-quickstart-grizzly2 \
-DarchetypeGroupId=org.glassfish.jersey.archetypes \
-DinteractiveMode=false \
-DgroupId=my.restful \
-DartifactId=my-first-service \
-Dpackage=my.restful \
-DarchetypeVersion=2.22.1
```

上述命令将创建一个标准的 Maven 工程。其中，interactiveMode=false 代表无需交互，archetypeVersion 指定原型的版本，这个版本与 Jersey 的版本一致。groupId、artifactId 和 package 分别定义了我们这个项目的组 ID 为 my.restful，工件 ID 为 my-first-service，包名为 my.restful。我们通过观察项目根目录下的 pom.xml，可以对应出上述命令参数与 Maven 坐标的关系。相关部分的示例如下。

```
<groupId>my.restful</groupId>
<artifactId>my-first-service</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>my-first-service</name>
```

2. 运行服务

Maven 工程建立好后，我们首先启动 REST 服务体验一下该项目的功能。进入项目的根目录，并执行如下命令构建和启动服务。

```
cd my-first-service
mvn package
mvn exec:java
...
Jersey app started with WADL available at http://localhost:8080/myapp/
application.wadl
Hit enter to stop it...
```

该命令启动了 REST 服务，端口是 8080，我们可以随时通过回车键停止这个服务。同时，该服务还提供了 WADL (详见 1.6 节)。通过访问 application.wadl，可以获取当前 REST

服务公布的接口。本例 WADL 的关键部分，示例如下。

```
<ns0:resources base="http://localhost:8080/myapp/">
  <ns0:resource path="myresource">
    <ns0:method id="getIt" name="GET">
      <ns0:response>
        <ns0:representation mediaType="text/plain" />
      </ns0:response>
    </ns0:method>
  </ns0:resource>
</ns0:resources>
```

这里定义了一个资源路径 myresource，在该路径下，定义了一个 GET 方法 getIt，表述类型为 text/plain。

3. 访问服务

我们使用 cURL（详见 1.8 节）来访问 REST 服务公布的 myresource 资源方法 getIt，示例如下。

```
curl http://localhost:8080/myapp/myresource
```

```
Got it!
```

HTTPie（读作 H-T-T-Pie）是和 cURL 类似的 CLI 工具，但交互上更人性化。我们使用 HTTPie 请求相同的资源地址，请求和响应信息如下。

```
http http://localhost:8080/myapp/myresource
```

```
HTTP/1.1 200 OK
Content-Length: 7
Content-Type: text/plain
Date: Sat, 14 Nov 2015 04:08:54 GMT
```

```
Got it!
```

响应信息的第一行包含了 HTTP 协议版本和状态码，接下来是部分 HTTP HEAD 信息，最后是 HTTP BODY 信息。cURL 携带 -i 或者 --include 参数可以得到相同的结果，示例如下。

```
curl -i http://localhost:8080/myapp/myresource
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Sat, 14 Nov 2015 04:08:54 GMT
Content-Length: 7
```

```
Got it!
```

要想获得更多的 cURL 请求响应信息，可以使用 -v 参数，示例如下。

```
curl -v http://localhost:8080/myapp/myresource
```



```

* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 8080 failed: Connection refused
*   Trying fe80::1...
* connect to fe80::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /myapp/myresource HTTP/1.1
> User-Agent: curl/7.38.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Sat, 14 Nov 2015 04:08:56 GMT
< Content-Length: 7
<
* Connection #0 to host localhost left intact
Got it!

```

4. 分析项目

完成了最初的体验后，我们来分析下面这个示例工程。首先，从启动服务的命令 `mvn exec:java` 入手。该命令实际调用了 `exec-maven-plugin` 插件中定义的一个值为 `java` 的 `goal`，用以触发 `mainClass` 中的 `main` 函数。本例的 `mainClass` 定义为 `my.restful.Main`。在 `pom.xml` 中，`exec` 插件完整定义如下。

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <executions>
    <execution>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <mainClass>my.restful.Main</mainClass>
  </configuration>
</plugin>

```

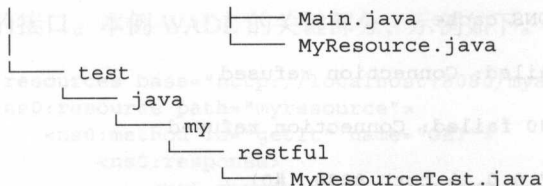
除了 `pom.xml` 和 `Main` 类，示例还包含哪些内容呢？我们可以使用如下命令查看。

tree .

```

.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── my
    │           └── restful

```



源代码中，还包括了资源类 `MyResource` 和它的单元测试类 `MyResourceTest`。

在资源类 `MyResource` 中，`@Path` 中定义了资源路径，`@GET` 中定义了 GET 方法 `getIt()`，`@Produces` 中定义了响应的类型为普通的字符串，示例如下。

```

@Path("myresource")
public class MyResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}

```

相应地，资源测试类 `MyResourceTest` 中实现了对 `getIt()` 方法的测试，示例如下。

```

@Test
public void testGetIt() {
    String responseMsg = target.path("myresource").request().get(String.class);
    assertEquals("Got it!", responseMsg);
}

```

在上述代码中，`target()`、`path()`、`request()` 和 `get()` 方法都是 `Jersey Client` 中定义的方法，这些方法组合在一起，形成了流式风格的 API。我们期待响应值为“Got it!”的字符串。

5. 单元测试

最后，我们使用如下命令执行单元测试。使用 IDE 可以直接通过图形界面单击对该方法的测试。

```
mvn test
```

```
...
```

```
-----
T E S T S
-----
```

```
Running my.restful.MyResourceTest
```

```
...
```

```
Results:
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

`jersey-quickstart-grizzly2` 原型提供的模板代码，使用了 `main` 函数，并在其中启动了 `Grizzly` 的 `HttpServer`。这是典型的 Java SE 形式的 REST 应用。更多情况下，我们希望得到的是一个可以以 `war` 包形式部署到 `Servlet` 容器的轻量级 Java EE 项目。接下来的示例就是

这样的 Web 形式的项目。

1.5.2 第一个 Servlet 容器服务

jersey-quickstart-webapp 原型会为我们生成 Servlet 容器服务。

1. 创建项目

使用如下命令创建名为 my-first-webapp 的 Web 项目。

```
mvn archetype:generate \
-DarchetypeArtifactId=jersey-quickstart-webapp \
-DarchetypeGroupId=org.glassfish.jersey.archetypes \
-DinteractiveMode=false \
-DgroupId=my.restful \
-DartifactId=my-first-webapp \
-Dpackage=my.restful \
-DarchetypeVersion=2.22.1
```

2. 运行服务

由于这是一个 Web 项目，没有 main 函数，我们必须将其部署到 Servlet 容器（比如 Tomcat、Jetty）中，才能将其运行。在开发阶段，我们无需真正将其部署，而是使用 Maven 插件这种更轻量级的方式启动服务。在 pom.xml 中，增加如下定义来添加插件。

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.3.5.v20151012</version>
</plugin>
```

有了插件，我们可以使用如下命令编译和启动服务，使用 Ctrl+C 停止服务。

```
mvn jetty:run
```

如果我们要对示例项目进行断点调试，应在服务启动前设置监听端口等信息。这里以 IntelliJ IDEA 所使用的 5050 端口为例，示例如下。

```
export MAVEN_OPTS="-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:
transport=dt_socket,address=5050,server=y,suspend=y"
mvn jetty:run
```

以这样的方式启动服务，需要 IDE 与之交互。过程是首先启动端口，然后 IDE 向该端口请求监听，服务启动并接收请求，在代码的某个断点处，服务会向该端口推送事件，IDE 在代码的断点处停留并高亮显示该行。

3. 访问服务

服务启动后，我们使用 HTTPie 请求资源地址，示例如下。

```
http http://localhost:8080/webapi/myresource
```

```
HTTP/1.1 200 OK
Content-Length: 7
Content-Type: text/plain
Date: Sat, 14 Nov 2015 08:00:03 GMT
Server: Jetty(9.3.5.v20151012)
```

```
Got it!
```

4. 分析项目

本例是一个标准的 Maven Web 工程。Web 的根目录默认名称为 webapp，默认的 Servlet 版本为 2.5，需要使用 WEB-INF/web.xml 文件来配置 REST 服务。我们通过 tree 命令得到完整的工程结构如下。

```
tree .
.
├── my-first-webapp.iml
├── pom.xml
├── src
│   └── main
│       ├── java
│       │   └── my
│       │       └── restful
│       │           └── MyResource.java
│       ├── resources
│       └── webapp
│           ├── WEB-INF
│           │   └── web.xml
│           └── index.jsp
```

5. 扩展项目

本例与前例提供的资源类和资源方法相同，我们在此基础上增加两个资源方法，分别用来新增和查询资源，示例如下。

```
private static ConcurrentHashMap<String, MyDomain> map=new ConcurrentHashMap<>();

@GET
@Path("/{key}")
@Produces(MediaType.APPLICATION_XML)
public MyDomain getMy(@PathParam("key") final String key) {
    final MyDomain myDomain = map.get(key);
    if (myDomain == null) {
        return new MyDomain();
    }
    return myDomain;
}

@POST
@Consumes(MediaType.APPLICATION_XML)
public void addMy(final MyDomain myDomain) {
```

```
map.put(myDomain.getName(), myDomain);
}
```

如上所示，POST 方法 addMy 用于接收并存储新增的表述，GET 方法 getMy 用于查询表述。MyDomain 类是基于 JAXB 的 POJO 类，用于表示 XML 格式的表述。

首先，我们通过如下命令，新增一条记录。

```
curl -X POST http://localhost:8080/webapi/myresource -d '<myDomain name="eric" value="feuyeux@gmail.com"/>' -H "Content-type:application/xml"
```

然后通过如下命令查询和验证新增记录的存在。

```
curl http://localhost:8080/webapi/myresource/eric
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><myDomain name="eric" value="feuyeux@gmail.com"/>
```

1.6 快速了解 Java REST 服务

1.6.1 REST 工程类型

在 REST 服务中，资源类是接收 REST 请求并完成响应的核心类，而资源类是由 REST 服务的“提供者”来调度的。这一概念类似其他框架中自定义的 Servlet 类，该类会将请求分派给指定的 Controller/Action 类来处理。本节将讲述 REST 中的这个提供者，即 JAX-RS2 中定义的 Application 以及 Servlet。

Application 类在 JAX-RS2 (JSR339, 详见参考资料) 标准中定义为 javax.ws.rs.core.Application，相当于 JAX-RS2 服务的入口。如果 REST 服务没有自定义 Application 的子类，容器将默认生成一个 javax.ws.rs.core.Application 类。

本节根据 JAX-RS2 规范第 2 章中对 REST 服务场景的定义，将 REST 服务分为四种类型，如图 1-1 所示。

图 1-1 将 JAX-RS2 标准中对 REST 服务的类型图形化，依据不同的条件分为了四种类型。

□ 类型一：当服务中没有 Application 子类时，容器会查找 Servlet 的子类来做入口，如果 Servlet 的子类也不存在，则 REST 服务类型为类型一，对应图 1-1 中的例 1。

□ 类型二：当服务中没有 Application 子类时，存在 Servlet 的子类，则 REST 服务类型为类型二，对应图 1-1 中的例 2。

□ 类型三：服务中定义了 Application 的子类，而且这个 Application 的子类使用了 @ApplicationPath 注解，则 REST 服务类型为类型三，对应图 1-1 中的例 3。

□ 类型四：如果服务中定义了 Application 的子类，但是这个 Application 的子类没有使

用 `@ApplicationPath` 注解，则 REST 服务类型为类型四，对应图 1-1 中的例 4。

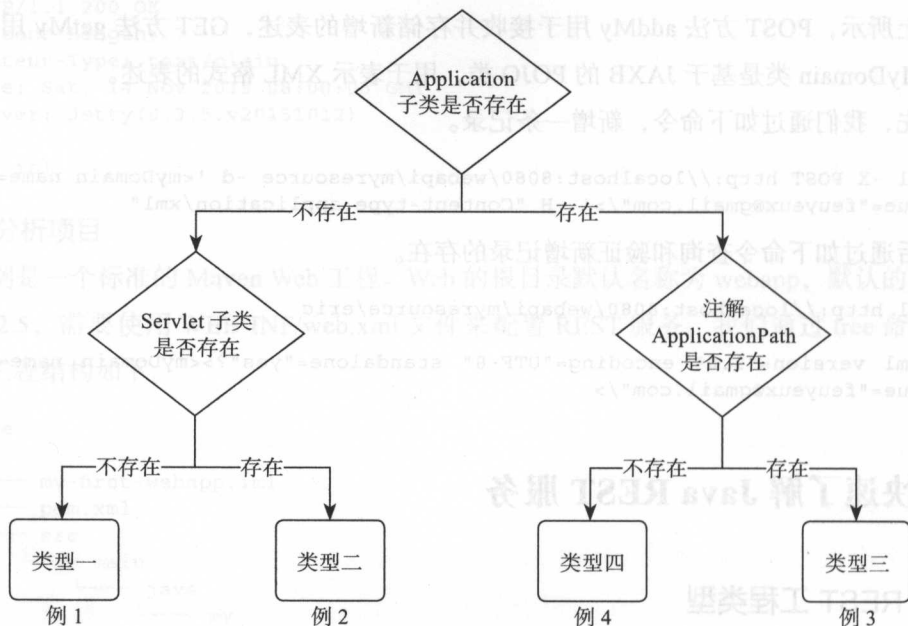


图 1-1 REST 工程类型示意图

上面提到的四个示例在下面的“阅读指南”中给出了源代码目录和 Github 下载地址，需要读者仔细体会示例之间的差异，以更好地理解和使用不同类型的 REST 服务。

1. REST 服务类型一

类型一对应的是图 1-1 中的例 1，相应的逻辑是服务中同时不存在 `Application` 的子类和 `Servlet` 子类。在 JAX-RS2 (JSR339) 中定义这种情况下应作如下处理：为 REST 服务动态生成一个名称为 `javax.ws.rs.core.Application` 的 `Servlet` 实例，并自动探测匹配资源。与此同时，需要根据 `Servlet` 的不同版本，在 `web.xml` 定义 REST 请求处理的 `Servlet` 为这个动态生成的 `Servlet`，并定义该 `Servlet` 对资源路径的匹配。在没有 `Application` 的子类存在的情况下，在 `web.xml` 中定义 `Servlet` 是必不可少的配置。

阅读指南

REST 服务类型一所对应的示例，即例 1 的源代码地址如下。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.1.myrest-servlet2-webxml>。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.2.myrest-servlet3-webxml>。

请使用 `mvn jetty:run` 启动服务，使用 `curl http://localhost:8080/webapi/myresource` 测试服务。

REST 服务类型一的示例包含两个小项目，分别对应 Servlet2 和 Servlet3 两种容器依赖场景。我们只须关注 Maven 配置文件 (pom.xml) 和 Web 服务配置文件 (web.xml) 的区别即可理解无 Application 子类情况下，如何实现基于 Servlet2 和 Servlet3 容器内的服务。

Servlet3 的最简配置示例代码如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java EE/web-app_3_0.xsd">
  <servlet>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
  </servlet>
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
  </servlet-mapping>
</web-app>
```

相对于 Servlet2 而言，在 Servlet3 中，servlet 的定义可以只包含 servlet-name。再次强调，Jersey 的 Servlet3 的容器支持包是 jersey-container-servlet。Servlet2 的最简配置示例代码如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java EE/web-app_2_5.xsd">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.example</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
  </servlet-mapping>
</web-app>
```

servlet 的定义包含 servlet-name 和 servlet-class，其初始化参数需要显示给出要加载的资源类所在的包名，可以看出 Servlet2 的支持包 jersey-container-servlet-core 不具备自动扫描资源类的功能。

2. REST 服务类型二

类型二对应的是图 1-1 中的例 2，相应的逻辑是不存在 Application 的子类但存在 Servlet 的子类。

阅读指南

REST 服务类型二所对应的示例，即例 2 的源代码地址如下。

□ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.3.myrest-subservlet>。

本例定义了 Servlet 子类 AirServlet，该类继承自 org.glassfish.jersey.servlet.ServletContainer 类，这是 Jersey2 中 Servlet 的基类，继承自 HttpServlet。AirServlet 类的代码示例如下。

```
@WebServlet(  
    initParams = @WebInitParam(  
        name = "jersey.config.server.provider.packages", value = "com.example"),  
        urlPatterns = "/webapi/*",  
        loadOnStartup = 1)  
    public class AirServlet extends ServletContainer {
```

AirServlet 使用了 WebServlet 注解来配置 Servlet 参数。包括初始化参数 initParams 中定义扫描的资源类所在的包名：com.example，Servlet 匹配的资源路径：urlPatterns="/webapi/*" 和启动时的加载标识：loadOnStartup=1。

例 2 是基于 Servlet3 容器的 REST 服务，使用了 WebServlet 注解和无 web.xml 等 Servlet3 引入而 Servlet2 没有的功能。在自定义 Servlet3.x 子类的场景下，web.xml 可以省略，但需要修改 Maven 的 maven-war-plugin 插件的配置，添加 failOnMissingWebXml 为 false，这样编译时才不会报错。Maven 配置文件中相关信息如下所示。

```
<plugin>  
    <groupId>org.apache.maven.plugins</groupId>  
    <artifactId>maven-war-plugin</artifactId>  
    <version>2.3</version>  
    <configuration>  
        <failOnMissingWebXml>false</failOnMissingWebXml>  
    </configuration>  
</plugin>  
<dependency>  
    <groupId>javax.servlet</groupId>  
    <artifactId>javax.servlet-api</artifactId>  
    <version>3.1.0</version>  
    <scope>provided</scope>  
</dependency>
```

3. REST 服务类型三

类型三对应的是图 1-1 中的例 3，相应的逻辑是存在 Application 的子类并且定义了 @ApplicationPath 注解。

阅读指南

REST 服务类型三所对应的示例，即例 3 的源代码地址如下。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.4.myrest-servlet3-application>。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.5.myrest-servlet2-rc>。

REST 服务类型三的示例包含两个小项目。其中，servlet2-rc 项目基于 Servlet2，AirResourceConfig 类继承自 Application 的子类 ResourceConfig 类；servlet3-application 项目基于 Servlet3，AirApplication 类继承自 Application 类。基于 Servlet2 的 REST 服务需要定义 web.xml（但内容可以是“空的”，即只有 web-app 的基本定义），基于 Servlet3 的 REST 服务可以省略此文件。AirApplication 类代码示例如下。

```
@ApplicationPath("/webapi/*")
public class AirApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(MyResource.class);
        return classes;
    }
}
```

AirApplication 类覆盖了 getClasses() 方法，注册了资源类 MyResource，这样在服务启动后，MyResource 类提供的资源路径将被映射到内存，以便请求处理时匹配相关的资源类和方法。AirResourceConfig 类代码示例如下。

```
@ApplicationPath("/webapi/*")
public class AirResourceConfig extends ResourceConfig {
    public AirResourceConfig() {
        packages("com.example");
    }
}
```

AirResourceConfig 类在构造子中提供了扫描包的全名，这样在服务启动后，com.example 包内资源类所提供的资源路径将被映射到内存。

4. REST 服务类型四

类型四对应的是图 1-1 中的例 4，相应的逻辑是一有二无：一有是存在 Application 的子类；二无是不存在 Servlet 子类、不存在或者不允许使用注解 @ApplicationPath。

阅读指南

REST 服务类型四所对应的示例，即例 4 的源代码地址如下。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.6.myrest-servlet2-application>。

❑ <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/1.6.7.myrest-servlet3-application>。

REST 服务类型四的示例包含两个小项目，演示了基于 Servlet2 和 Servlet3 两个版本的 REST 服务，其差异仅此而已，关于差异性配置前面的例子已经讲过，不再冗述。如下以 servlet3-application 为例说明。AirApplication 类是 Application 的子类，代码示例如下。

```
public class AirApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(MyResource.class);
        return classes;
    }
}
```

代码和类型三的示例相仿，但是该类没有定义 @ApplicationPath 注解，因此我们需要在 web.xml 中配置 Servlet 和映射资源路径，代码示例如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/J2EE" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
J2EE http://java.sun.com/xml/ns/J2EE/web-app_3_0.xsd" version="3.0">
    <servlet>
        <servlet-name>com.example.AirApplication</servlet-name>
    </servlet>
    <servlet-mapping>
        <servlet-name>com.example.AirApplication</servlet-name>
        <url-pattern>/webapi/*</url-pattern>
    </servlet-mapping>
</web-app>
```

在 servlet-name 中使用自定义的 Application 子类 com.example.AirApplication 的全名作为 Servlet 名称，并在 url-pattern 中映射资源路径。

1.6.2 REST 应用描述

在明白如何创建和部署各种类型的 REST 服务后，我们来了解一下部署好的 REST 服务中一个特殊的成员，REST 应用的描述：以 XML 格式展示当前 REST 环境中所提供的 REST 服务接口。这种 XML 格式的描述就是 WADL（Web Application Description Language，Web 应用描述语言）。

WADL 是用来描述基于 HTTP 协议的 REST 式 Web 服务部署情况的。它采用 XML 格式，支持多种数据类型的描述。WADL 由 Sun 公司提出，尚未成为 W3C 或者 OASIS 的标准，JAX-RS 标准中并没有关于 WADL 的定义和说明。Jersey 作为 JAX-RS2 的参考实现默认支持服务的 WADL。通过浏览器访问“服务根路径/application.wadl”即可打开该服务的 WADL 内容。相对于 REST 服务，WSDL 更为人们所熟知，WSDL 是 RPC 风格的基于 SOAP 的 Web 服务的描述语言。两者缩写类似而且都使用 XML 格式，此外共性不多。

1. 应用的描述

以 REST 服务类型四的示例项目 1.6.7.myrest-servlet3-application 为例，该应用的 WADL 路径如下：<http://localhost:8080/myrest-servlet3-application/webapi/application.wadl>。

通过浏览器访问该路径，可以一览 WADL 的 schema 结构。WADL 的最外层标签是 application，代表应用。然后自上而下分别是 doc、grammars 和 resources。resources 是应用提供的资源集合，里面至少包含 application.wadl，以及应用中包含的资源描述，比如本例的资源信息描述在资源路径 myresource 之内，如下所示。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application>
  <doc jersey:generatedBy="Jersey: 2.3 2013-09-20 13:59:07"/>
  <grammars/>
  <resources
    base="http://localhost:8080/myrest-servlet3-application/webapi/">
    <resource path="myresource">...</resource>
    <resource path="application.wadl">...</resource>
  </resources>
</application>
```

2. 资源的描述

可以展开 myresource 来查看具体某个方法的 WADL，也可以通过发送一条请求并定义请求头信息来获取。以 cURL（详见 1.8 节）为例，命令如下。

```
curl -X OPTIONS -H "Allow: application/vnd.sun.wadl+xml" -v http://localhost:
8080/myrest-servlet3-application/webapi/myresource
```

myrest-servlet3-application 提供的资源接口，对照服务器返回的 XML，可以更清晰地理解 WADL 的内容。其 WADL 内容如下。

```
<resource path="myresource">
  <method id="getIt" name="GET">
    <response>
      <representation mediaType="text/plain"/>
    </response>
  </method>
  <method id="apply" name="OPTIONS">
    <request>
      <representation mediaType="**/*"/>
    </request>
    <response>
      <representation mediaType="application/vnd.sun.wadl+xml"/>
    </response>
  </method>
  <method id="apply" name="OPTIONS">
    <request>
      <representation mediaType="**/*"/>
    </request>
    <response>
```

```

        <representation mediaType="text/plain"/>
    </response>
</method>
<method id="apply" name="OPTIONS">
    <request>
        <representation mediaType="*/*/">
    </request>
    <response>
        <representation mediaType="*/*/">
    </response>
    </method>
</resource>

```

在这段代码中，公布了四个方法。其中，getIt 方法代码如下。其他三个 OPTIONS 请求方法是 Jersey 默认实现的，用以描述 getIt 方法，分别返回 text/plain 类型，*/* 类型和 application/vnd.sun.wadl+xml 类型。

```

@GET
@Produces(MediaType.TEXT_PLAIN)
public String getIt() {
    return "Got it!";
}

```

getIt 方法定义为 GET 请求方法，@Produces 中定义的媒体类型是 MediaType.TEXT_PLAIN，即响应过程中生产的数据，其表述性状态以 text/plain 媒体类型转移。

3. WADL 的配置

上述 OPTIONS 请求方法的实现是 Jersey 默认支持的，如果读者不希望在 REST 服务中让 Jersey 自动生成，可以通过配置 jersey.config.server.wadl.disableWadl=true 来实现。代码示例如下。

```

public class AirApplication extends ResourceConfig {
    public AirApplication() {
        property(ServerProperties.WADL_FEATURE_DISABLE, true);
        packages("com.example.resource");
    }
}

```

在构造函数中，我们通过定义 ServerProperties.WADL_FEATURE_DISABLE 属性为 true 以实现去除 WADL 自动生成的功能。或者，可以通过修改 Web 配置文件中 servlet 启动参数来实现，代码示例如下。

```

<servlet>
    <servlet-name>com.example.AirApplication</servlet-name>
    <init-param>
        <param-name>jersey.config.server.wadl.disableWadl</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>

```

配置文件中定义了启动参数 `jersey.config.server.wadl.disableWadl`，其值定义为 `true`，以实现去除 WADL 自动生成的功能。

1.7 Java 领域的其他 REST 实现

Java 领域存在很多 REST 实现，我们以是否遵循 JAX-RS 标准，将它们分为两组。前者是 JAX-RS 标准参考实现之外的厂商实现，后者要么是因为出现较 JAX-RS 标准早，要么干脆跳出了 JAX-RS 标准的定义，以自身框架一致性为目标，实现了一套独有的对 REST 开发的支持。本节将概括性地介绍这些实现工具，以便读者有所对比和选择。

1.7.1 JAX-RS 的其他实现

JAX-RS 标准发布后，诸多厂商推出了自己的基于 JAX-RS 标准的实现。其中最有影响力的当属来自 JBoss 社区的 RESTEasy 和来自 Apache 社区的 CXF。本节将简述这两个项目。如果读者的项目确实和它们结合得紧密，Jersey 未必是最佳选择，读者尽可拥抱这两个基于 JAX-RS 标准的项目。

1. JBoss 的 RESTEasy

RESTEasy 是 JBoss 社区提供的 JAX-RS 项目。JBoss 这一名词目前已经不再代表 Java EE 容器，曾经的 JBoss 已经更名为 WildFly。现在，JBoss 特指 RedHat 公司旗下的开源社区。RESTEasy 自 2009 年 1 月第一个 GA 版本以来，发展到 3.0.x，从版本 3.0.0.Final 开始支持 JAX-RS2.0。RESTEasy 的基本信息如下。

- ❑ 官方网站：<http://resteasy.jboss.org>。
- ❑ 官方文档：提供单页面 HTML、按章节 HTML 和 PDF 3 种格式，可以按照阅读习惯选择。地址为 <http://resteasy.jboss.org/docs.html>。
- ❑ 源代码：由 GitHub 托管，地址为 <https://github.com/resteasy/Resteasy>。
- ❑ 缺陷管理：地址为 <https://issues.jboss.org/projects/RESTEASY>，隶属于 JBoss 开发者社区的 JIRA 系统。
- ❑ 项目下载：由 sourceforge 托管，地址为 <http://sourceforge.net/projects/resteasy/files/Resteasy%20JAX-RS>。维护的最后一个版本是 3.0.9.Final，日期为 2014-09-17，而目前 RESTEasy 的最新版本为 3.0.13.Final。

(1) 快速开始

最快速地了解一个新框架的办法，一定是从官方提供的示例开始。我们首先迁出 RESTEasy 的源代码，然后进入示例目录，选择最易上手的示例项目。这里以 `resteasy-`

springMVC 为例，示例操作如下。

```
git clone https://github.com/reteasy/Resteasy.git
cd Resteasy/jaxrs/examples/reteasy-springMVC
mvn clean install
mvn jetty:run
```

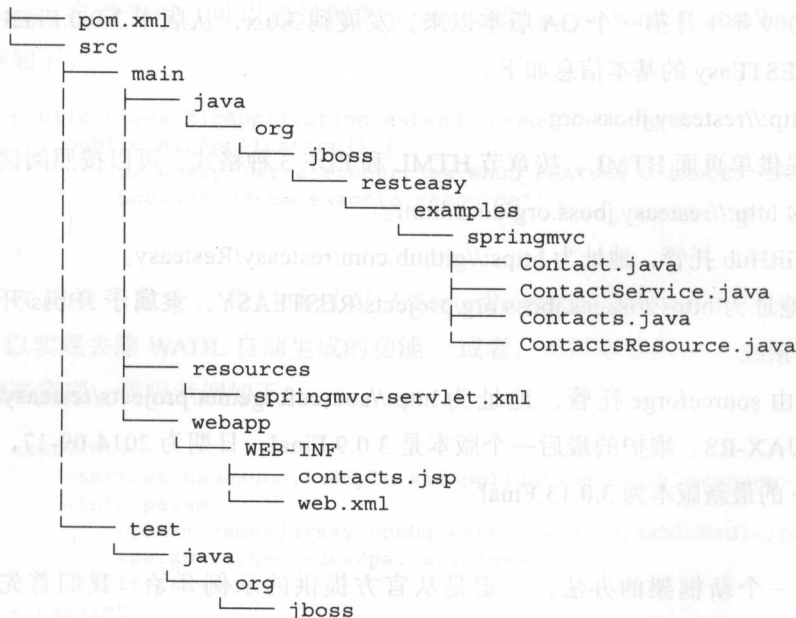
上述操作分别执行了代码迁出、进入示例目录、构建示例项目和启动示例服务。当服务启动完毕后，我们可以在浏览器的地址栏输入 `http://localhost:8080/contacts`，在页面中输入测试数据并提交表单，这里提交的是 Eric 和 Han。测试数据会被保存在 REST 服务的内存中，可以通过如下 GET 请求获取服务端保存的数据，示例结果如下。

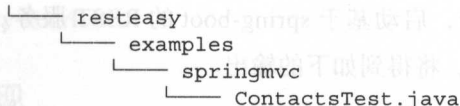
```
curl http://localhost:8080/contacts/data/Han
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<contact>
  <firstName>Eric</firstName>
  <lastName>Han</lastName>
</contact>
```

从上面的测试结果中，我们得到了 lastName 为 Han，XML 格式的 contact 资源数据。

(2) 简单分析

这个示例项目是非常典型的 JAX-RS 实践。观察如下所示的目录树，源代码路径中包含了 4 个类，其中 Contact 和 Contacts 是实体类，ContactService 是服务类，ContactsResource 是 REST 服务的资源类，负责定义 REST 接口。单元测试类 ContactsTest 是对 ContactsResource 功能的全面测试。





示例项目所依赖的包简述如下。

❑ org.jboss.resteasy

- **resteasy-spring**: 支持 RESTEasy 与 Spring 集成, 底层依赖 **resteasy-jaxrs**。**resteasy-jaxrs** 是 RESTEasy 的 JAX-RS 核心包。
- **resteasy-client**: RESTEasy 客户端包。
- **resteasy-jaxb-provider**: RESTEasy 的 XML 处理包。

❑ org.jboss.spec.javax.servlet

- **jboss-servlet-api_3.1_spec**: RESTEasy 的 JBoss Servlet 实现。

❑ org.springframework

- **spring-webmvc**: Spring MVC 包。

2. Apache 的 CXF

CXF 是 Apache 开源社区提供的 JAX-RS 项目, CXF 的名称是由 Celtix 项目和 XFire 项目合并而来。其中 Celtix 由 IONA Technologies 开发, XFire 来自 Codehaus。CXF 是 JAX-WS 的著名实现, 同时实现了 JAX-RS, 从版本 2.7.0 开始几乎全面支持 JAX-RS2.0 全部特性。从版本 3.0.0 开始实现 JAX-RS2 客户端 API。CXF 的基本信息如下。

❑ 官方网站: <http://cxf.apache.org>。

❑ 官方文档: <http://cxf.apache.org/docs/jax-rs.html>。

❑ 项目下载: Apache CXF 当前版本是 3.1.3。下载地址为 <http://cxf.apache.org/download.html>。

❑ 源代码: 由 Apache 的 GIT 服务器托管, 地址为 <https://git-wip-us.apache.org/repos/asf?p=cxf.git>; 另有一个自动镜像由 GitHub 托管, 地址为 <https://github.com/apache/cxf>。

❑ 邮件列表: <http://cxf.apache.org/mailling-lists.html>。

(1) 快速开始

我们依然从官方提供的示例开始。首先迁出 CXF 的源代码, 然后进入示例目录, 这里选择的示例项目为 **jaxrs_spring_boot**, 示例操作如下。

```
git clone https://git-wip-us.apache.org/repos/asf/cxf.git
git clone https://github.com/apache/cxf.git
cd cxf/distribution/src/main/release/samples/jax_rs/jaxrs_spring_boot
mvn spring-boot:run
```

可以从上述的前 2 行中选择一个地址, 将 CXF 源代码迁出。进入 **jaxrs_spring_boot** 目

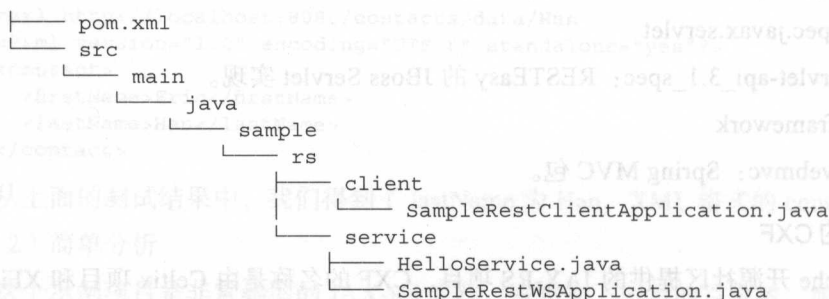
录，然后执行 maven 命令，启动基于 spring-boot 的 REST 服务。服务启动后，我们在另一个终端中使用 cURL 测试，将得到如下的输出。

```
curl " :8080/services/helloservice/sayHello/ApacheCxfUser"
```

```
Hello ApacheCxfUser, Welcome to CXF RS Spring Boot World!!!
```

(2) 简单分析

该示例项目也是典型的 JAX-RS 实践。观察如下所示的目录树，HelloService 是资源类，SampleRestWSApplication 是服务器端包含 main 方法的入口类，SampleRestClientApplication 是客户端包含 main 方法的入口类。



示例项目所依赖的包简述如下。

❑ spring-boot: Spring-boot 启动包

- org.springframework.boot:spring-boot-starter
- org.springframework.boot:spring-boot-starter-web

❑ cxf: CXF 包

- org.apache.cxf:cxf-rt-frontend-jaxrs
- org.apache.cxf:cxf-rt-rs-client
- org.apache.cxf:cxf-rt-transports-http
- org.apache.cxf:cxf-rt-rs-service-description

❑ jackson: JSON 处理包

- com.fasterxml.jackson.core:jackson-core
- com.fasterxml.jackson.core:jackson-databind
- com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider
- com.fasterxml.jackson.core:jackson-annotations

❑ jetty: Servlet 容器 Jetty 包

- org.eclipse.jetty:jetty-servlet
- org.eclipse.jetty:jetty-webapp

- org.eclipse.jetty:jetty-servlets

1.7.2 其他的 REST 实现

本节将介绍 Java 领域，没有遵循 JAX-RS 规范的 REST 式 Web 服务开发工具，包括 Restlet、LinkedIn 的 Rest.li 以及 Spring MVC。

1. Restlet 项目

Restlet 是一款遵从 REST 风格的、基于 Java 平台的轻量级框架，当前版本为 2.3。Restlet 许可为免费开源，提供 REST 开发的完整支持。Restlet 的基本信息如下。

- 官方网站: <http://restlet.org>。
- 源代码: 由 GitHub 托管，地址为 <https://github.com/restlet/restlet-framework-java>。
- StackOverflow: <http://stackoverflow.com/questions/tagged/restlet>。
- 学习指南文档: <http://restlet.org/learn/tutorial>。

(1) 快速开始

为了快速了解 Restlet 的使用，我们从一个现成的 Restlet 示例项目开始学习，示例如下。

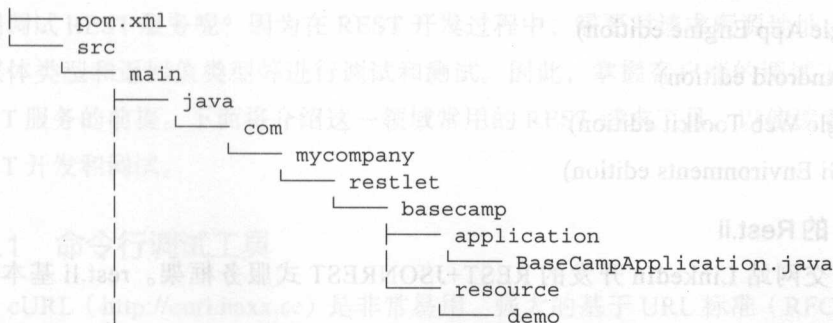
```
wget http://restlet.com/technical-resources/restlet-framework/archives/
examples/maven-spring/2.3/restlet-basecamp.zip
unzip restlet-basecamp.zip
cd restlet-basecamp
mvn jetty:run-war
```

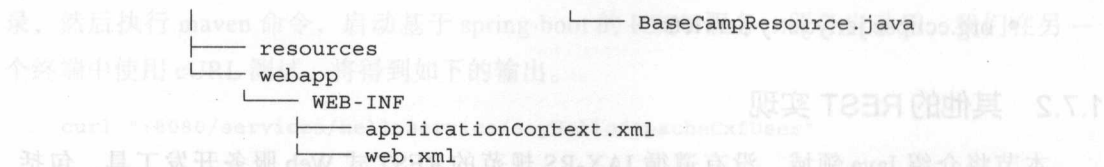
上述操作分别执行了下载示例包、解压缩、进入示例项目目录以及启动服务。服务启动后，我们使用 cURL 测试，将得到如下的输出。

```
curl "8080/basecamp/hello"
Hello World!
```

(2) 简单分析

Restlet 使用配置文件声明服务，我们观察如下所示的目录树，applicationContext.xml 是配置文件，BaseCampApplication 是 REST 服务入口类，BaseCampResource 是资源类。





配置文件 `applicationContext.xml` 展示如下，首先需要定义一个 Spring 组件 `basecampComponent`，它的 `defaultTarget` 属性指向 `BaseCampApplication` 的实例 `basecampApplication`。`basecampApplication` 的根路径路由指向 `SpringBeanRouter` 的实例。最后，`/hello` 路径指向 `BaseCampResource` 的实例。

```

<bean id="basecampComponent" class="org.restlet.ext.spring.SpringComponent">
  <property name="defaultTarget" ref="basecampApplication" />
</bean>

<bean id="basecampApplication" class="com.mycompany.restlet.basecamp.application.
BaseCampApplication">
  <property name="root" ref="router" />
</bean>

<!-- Define the router -->
<bean name="router" class="org.restlet.ext.spring.SpringBeanRouter" />

<!-- Define all the routes -->
<bean name="/hello" class="com.mycompany.restlet.basecamp.resource.demo.
BaseCampResource" scope="prototype" autowire="byName" />
  
```

示例项目所依赖的包简述如下。

- ☐ `org.restlet.jee:org.restlet:jar:2.0.1`
- ☐ `org.restlet.jee:org.restlet.ext.servlet:jar:2.0.1`
- ☐ `org.restlet.jee:org.restlet.ext.spring:jar:2.0.1`
 - `org.springframework:spring-webmvc:jar:3.0.1.RELEASE`

其中，`jee` 代表支持 Java EE 平台的包。此外，Restlet 支持的平台体系如下。

- ☐ `jse` (Java SE edition)
- ☐ `jee` (Java EE edition)
- ☐ `gae` (Google App Engine edition)
- ☐ `android` (Android edition)
- ☐ `gwt` (Google Web Toolkit edition)
- ☐ `osgi` (OSGi Environments edition)

2. LinkedIn 的 Rest.li

Rest.li 是社交网站 LinkedIn 开发的 REST+JSONREST 式服务框架。rest.li 基本信息如下。

❑ 官方地址: rest.li (<http://rest.li>)。

❑ 源代码: 由 GitHub 托管, 地址为 <https://github.com/linkedin/rest.li>。

❑ wiki: <https://github.com/linkedin/rest.li/wiki>。

- 快速开始教学文档: <https://github.com/linkedin/rest.li/wiki/Quickstart:-A-Tutorial-Introduction-to-Rest.li>。

快速开始

Rest.li 的项目是使用 Gradle 构建的, 完整的代码迁出和构建示例如下。

```
git clone https://github.com/linkedin/rest.li.git
cd rest.li/examples/spring-server
gradle build
gradle JettyRunWar
curl -v http://localhost:8080/fortunes/1
```

3. Spring MVC 项目

Spring 框架使用 Gradle 构建和管理项目, 使用 GIT 管理源代码, 地址为 <https://github.com/spring-projects/spring-framework>, 其中 MVC 模块位于 `spring-framework/spring-webmvc` 目录下。

Spring 从版本 3.0 开始提供了对 REST 式应用开发的支持, 但 Spring 目前并没有也没必要推出一个实现 JAX-RS 标准的模块。MVC 模块提供的 REST 功能并没有采用 JAX-RS 提出的标准。本质上, Spring MVC 控制流程是使用 Controller 处理 Model 在某种动词性的业务逻辑操作, 而 JAX-RS 的控制流程是使用资源类 Resource 处理名词性的资源表述。

在云概念和微服务大行其道的今天, Spring Boot 成为依赖 `spring-framework` 项目的事实标准。本书的第 7 章将讲述 Spring Boot 在 REST 服务领域中的影响力及实践。

1.8 REST 调试工具

在上节中, 我们不但领略了 REST 请求处理流程, 还对 IDE 中设置断点、观察服务器端运行时变量有了了解。本节将讲述如何在客户端对 REST 服务进行调试。为何需要在客户端调试 REST 服务呢? 因为在 REST 开发过程中, 需要对请求资源地址、资源所支持的数据媒体类型和返回值类型等进行调试和测试。因此, 掌握客户端的调试工具是开发优秀的 REST 服务的前提。下面将介绍这一领域常用的 REST 请求工具, 以使读者更进一步地熟悉 REST 开发和调试。

1.8.1 命令行调试工具

cURL (<http://curl.haxx.se>) 是非常易用、强大的基于 URL 标准 (RFC 3986) 的命令行

工具,通过命令行即可完成多种协议(比如 HTTP)的请求,并可以将请求的响应信息输出在终端/控制台上,因此对于调试和测试 REST 请求非常方便。

HTTPie (<http://httpie.org>) 是和 cURL 非常类似的命令行工具,相比 cURL 有更良好的用户体验。

命令行工具的优点是简单方便,缺点是没有图形化界面。下面将介绍几款基于浏览器的扩展插件作为 REST 客户端调试工具的使用情况。

1.8.2 基于浏览器的图形化调试插件

cURL 功能强大、易于在自动化脚本中使用,但 cURL 的每个请求都要通过码字来完成、没有图形界面的特点并不适于所有读者。下面将介绍几种基于浏览器的图形化调试插件,以方便读者在开发和测试 REST 服务时选择使用。基于 Chrome 浏览器的 REST 插件有很多,本节将介绍其中的 3 种。

1. Simple REST Client 插件

Simple REST Client 插件是基于 Chrome 浏览器的扩展,安装该插件后 Chrome 窗口的右上方会出现该插件的图标,以方便使用。该项目的地址是 <https://github.com/jeremys/Simple-REST-Client-Chrome-Extension>, 插件的下载地址是 <http://chrome.google.com/extensions/detail/fhjcajmcblldhlcimfajhfbgofnpejmb>。Simple REST Client 插件的界面如图 1-2 所示。

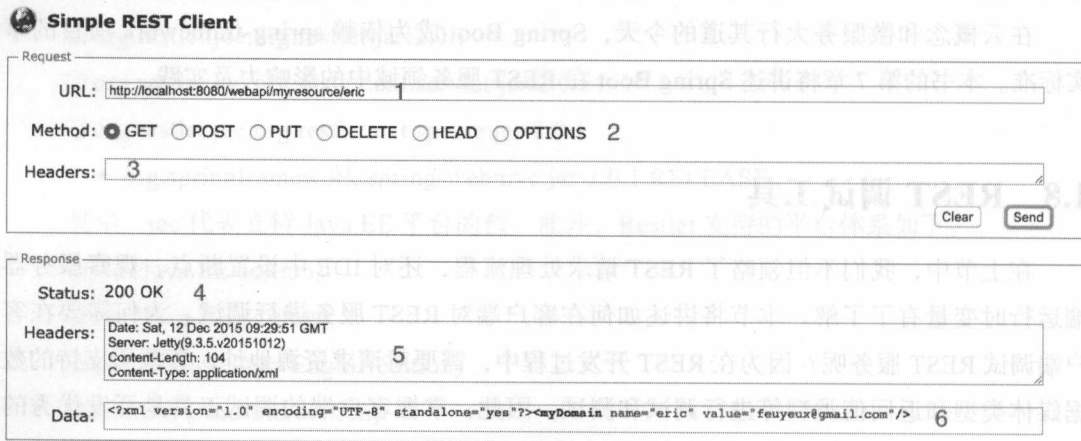


图 1-2 Simple REST Client 插件示意图

Simple REST Client 插件的特点是简单易用,其界面分为请求信息录入和响应信息展示上下两部分。录入部分包括 URL、HTTP 请求方法和请求头 3 部分,见图 1-2 中上方的数字标识 1 ~ 3。其中,HTTP 请求方法支持 HTTP 的标准方法 GET、POST、PUT、DELETE、

HEAD 和 OPTIONS, Headers 部分需要完全手工输入。响应信息部分包括响应状态、响应头和响应实体 3 部分。其中, Headers 部分展示 HTTP 请求交互的响应头信息, Data 中展示的是响应实体信息, 语法高亮显示, 见图 1-2 中下方的数字标识 4 ~ 6。Simple REST Client 插件总体上是麻雀虽小, 五脏俱全, 但功能相比后面要讲的插件不够强大。

2. Advance REST Client 插件

Advance REST Client 可以看作是 Simple REST Client 的增强版。该项目的地址是 <https://code.google.com/p/chrome-rest-client>, 插件的下载地址是 <https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfjeloo>。界面如图 1-3 所示。

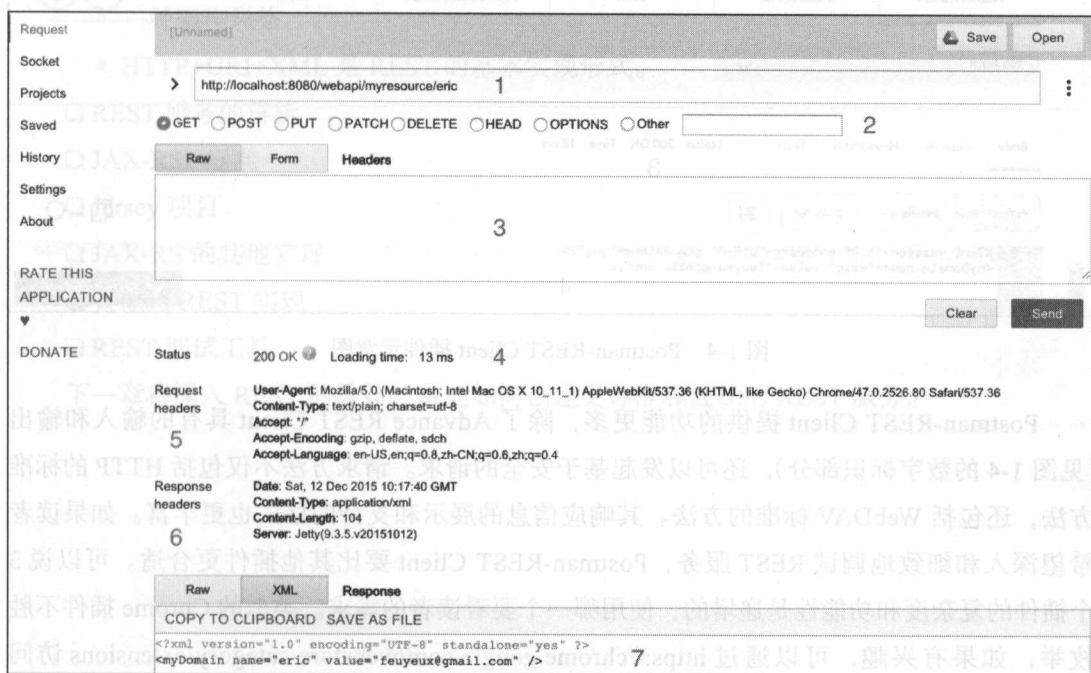


图 1-3 Advance REST Client 插件示意图

Advance REST Client 提供更为丰富的功能, 除了 Simple REST Client 插件具备的输入和输出 (见图 1-3 中的数字标识), Advance REST Client 还支持带参数的请求和提交表单等更完整的请求功能。数据格式上, 支持原生的格式 (Raw)、XML 格式的响应 (Response) 信息。Advance REST Client 支持对请求地址的保存和对最近使用地址的记忆, 如果调试中需要多次测试同一个资源地址, 可以将其保存下来为以后使用; 而多个这样的地址也可以按照项目分别保存, 方便区分使用。

3. Postman-REST Client 插件

Postman-REST Client 是基于 Simple REST Client 源代码编写的专门针对 REST 的插件。该项目的地址是 <https://github.com/a85/POSTMan-Chrome-Extension>，插件的下载地址是 <http://www.getpostman.com>。界面如图 1-4 所示。

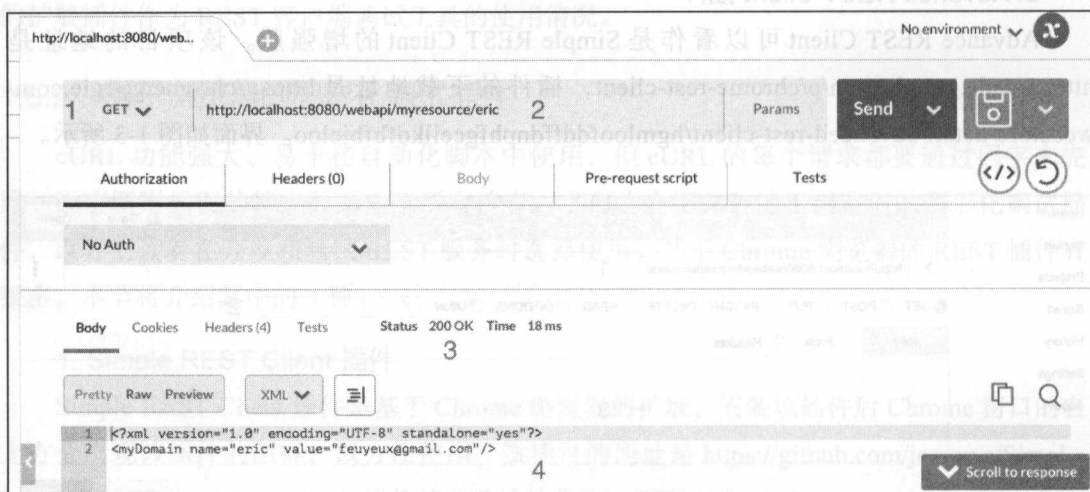


图 1-4 Postman-REST Client 插件示意图

Postman-REST Client 提供的功能更多，除了 Advance REST Client 具有的输入和输出（见图 1-4 的数字标识部分），还可以发起基于安全的请求。请求方法不仅包括 HTTP 的标准方法，还包括 WebDAV 标准的方法。其响应信息的展示和支持的格式也更丰富。如果读者希望深入和细致地调试 REST 服务，Postman-REST Client 要比其他插件更合适。可以说 3 个插件的复杂度和功能性是递增的，使用哪一个要看读者的需求。类似的 Chrome 插件不胜枚举，如果有兴趣，可以通过 <https://chrome.google.com/webstore/category/extensions> 访问 Chrome 的网上商店，搜索更多的 Chrome 插件。

4. Firefox 插件

相对于 Chrome 浏览器，Firefox 的 REST 插件功能类似，其中常用的插件有 REST-Easy 和 RESTClient。REST-Easy 的项目地址是 <https://github.com/nathan-osman/REST-Easy>，RESTClient 的项目地址是 <http://restclient.net>。类似的 Firefox 插件很多，读者如果有兴趣，可以在 Firefox 浏览器中录入 `about:addons` 进入 Firefox 的扩展，搜索更多的相关插件。

1.9 本章小结

本章主要讲述了 REST 服务的概念和实战。先后解读了 REST、REST 服务、JAX-RS2 标准中的重要概念，对 JAX-RS2 的参考实现项目 Jersey 进行了简单而全面的概述。随后讲述了如何快速创建 REST 应用和 REST 服务，介绍了基于 JAX-RS2 标准的其他项目和其他非 JAX-RS2 标准的、著名的 Java 项目。

通过阅读本章，读者可以清楚地掌握 Java 领域开发 REST 服务中的基本概念。

本章主要的知识点如下。

□ REST 是什么

- 一种架构风格。
- HTTP+URI+XML 是 REST 的基本实现形式。

□ REST 服务的辨析

□ JAX-RS2 标准

□ Jersey 项目

□ JAX-RS 的其他实现

□ 其他的 REST 实现

□ REST 调试工具

下一章将深入 REST 的设计，讲述如何创建更标准和健壮的 REST 服务。

REST API 设计

设计和开发 REST 式的 Web 服务除了要掌握 JAX-RS2 标准,还要对统一接口、资源定位以及请求处理过程中 REST 风格的传输数据的格式、响应信息等有良好的认知。此外,设计良好的 REST API 应当对内容协商、资源地址信息(link)有良好的支持。

本章将详细讨论这些技术细节。

2.1 统一接口

REST 服务和 RPC 服务在接口定义上的区别是:REST 使用 HTTP 协议的通用方法作为统一接口的标准词汇,REST 服务所提供的方法信息都在 HTTP 方法里,而 RPC 服务所提供的方法信息在 SOAP/HTTP 信封里(其封装的格式通常是 HTTP 或 SOAP),每一个 RPC 式的 Web 服务都会公布一套符合自己商业逻辑的方法词汇。

阅读指南

本节示例源代码地址:<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包:com.example.annotation.method。

每一种 HTTP 请求方法都可以从安全性和幂等性两方面考虑,这对正确理解 HTTP 请求方法和设计统一接口具有决定性的意义。换句话说,要定义严谨的 REST 统一接口,就需

要真正理解 HTTP 方法的安全性和幂等性。

安全性是指外系统对该接口的访问，不会使服务器端资源的状态发生改变；幂等性 (idempotence) 是指外系统对同一 REST 接口的多次访问，得到的资源状态是相同的。

阅读指南

这里讨论的安全性对应的英文是 **Safety** 而不是 **Security**，系统安全请参考第 10 章。以下，将从 REST 统一接口的定义角度，逐个讲述 HTTP 方法。

2.1.1 GET 方法

REST 使用 HTTP 的 GET 方法获取服务提供的资源。GET 方法是只读的，那么它是幂等和安全的吗？答案马上揭晓。

1. 幂等性和安全性

HTTP 的 GET 方法用于读取资源。GET 方法是幂等的，因为读取同一个资源，总是得到相同的数据。GET 方法也是安全的，因为读取资源不会对其状态做改动。JAX-RS2 定义了 `@GET` 注解对资源方法定义，使得该方法用于处理 GET 请求。

值得注意的是，虽然 GET 方法的特性是幂等和安全的，但这并不意味着任何一个定义为处理 GET 请求的方法都是幂等和安全的。换句话说，设计不良的 API 有可能违背 GET 的特性，将一个不该是 GET 的方法定义为之。

举个例子，在系统 B 中设计一个 REST 的 API，在客户端调用时读取系统 A 中 x 类型的数据，然后将 A.x 与系统 B 内的 y 类型数据做比较，如果两个集合的内容、最后更新时间上有不同，需要执行同步数据，即将 A.x 追加或者更新到 B.y 中。最后，将同步结果信息返回给向系统 B 发起请求的客户端，如图 2-1 所示。

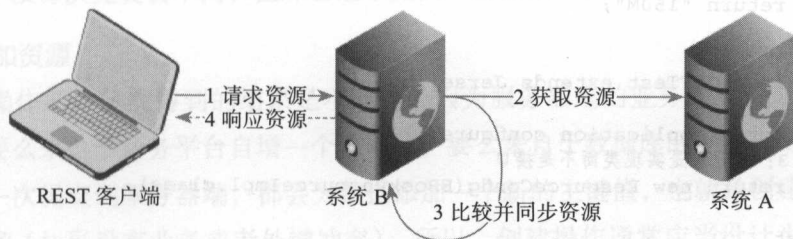


图 2-1 请求资源示意图

从图 2-1 左侧部分乍看上去，这是一个获取同步信息的 API，因此这个 API 的设计应该

使用 GET 请求方法。但是，稍作分析后即可知道该场景并不具备使用 GET 的基本条件。因为同步过程中对系统 B 内的资源有写操作的可能，因此不具备安全性；而写的内容又不是每次相同，因此不具有幂等性。所以，这个例子应该定义的正确请求方法是 POST。

2. 资源方法命名

不妨一起探讨一下图 2-1 中的同步信息的 API 该如何命名？既然是同步功能，那就以 sync 一类的字根作为前缀，这样所有的同步 API 都具有相同的开头，字迹也很工整。遗憾的是，这样的设计并不符合 REST 风格。笔者的理解是，从字面上看有两个问题。第一，sync 字根具有非名词性的含义，从 ROA 角度上看，sync 是 RPC 风格的命名：动词、自定义方法名称。第二，这样命名后，资源名称从一个主语变成了宾语，从 ROA 角度上看，面向的不再是资源，而是要执行的动作。

因此，标准的命名方式应该是单数的同步操作以资源名称命名；批量的同步操作以资源名称的复数名称命名。比如这个 API 是用于同步设备的，那么命名可以使用 device 和 devices。如果担心与普通查询业务资源地址混淆，可以在资源路径中增加查询或者路径参数，比如 device/id=1&source=a_b、device/b/a/ 等。

3. 抽象层注解资源

JAX-RS2 的 HTTP 方法注解可以定义在接口和 POJO 中，置于接口中的方法名更具抽象性和通用性。示例代码如下。

```
@Path("book")
public interface BookResource {
    // 关注点 1: GET 注解从抽象类上移到接口
    @GET
    public String getWeight();
}

public class EBookResourceImpl implements BookResource {
    // 关注点 2: 实现类无须 GET 注解
    @Override
    public String getWeight() {
        return "150M";
    }
}

public class GETTest extends JerseyTest {
    @Override
    protected Application configure() {
        // 关注点 3: 加载的是实现类而不是接口
        return new ResourceConfig(EBookResourceImpl.class);
    }
    @Test
    public void testGet() {
        Response response = target("book").request().get();
        Assert.assertEquals("150M", response.readEntity(String.class));
    }
}
```

在这段代码中，资源接口 `BookResource` 定义了一个 GET 方法 `getWeight()`，这个方法使用了 HTTP 方法注解 `@GET`，见关注点 1。资源接口 `BookResource` 的实现类 `EBookResourceImpl` 实现了 `getWeight()` 方法，但没有再次使用 `@GET` 注解。也就是说，在接口中抽象地定义了资源的请求方法类型后，其全部实现类都无须再定义。这使得编码更整洁和抽象，见关注点 2。最后，需要注意的是，在测试类 `GETTest` 中注册的是实现类 `EBookResourceImpl` 类型而不是接口 `BookResource` 类型，见关注点 3。

另外，我们一并介绍下 HEAD 方法和 OPTIONS 方法。HEAD 方法和 GET 方法相似，只是服务器端的返回值不包括 HTTP 实体。因此，HEAD 方法是安全的和幂等的。JAX-RS2 定义了 `@HEAD` 注解来定义相关资源方法。OPTIONS 方法和 GET 方法相似，是安全的和幂等的。OPTIONS 用于读取资源所支持的（Allow）所有 HTTP 请求方法。JAX-RS2 定义了 `@OPTIONS` 注解来定义相关资源方法。

2.1.2 PUT 方法

PUT 方法是一种写操作的 HTTP 请求。REST 使用 HTTP 的 PUT 方法更新或添加资源。下面讲解一下 PUT 方法的作用和操作时的媒体类型。

1. 更新资源

因为 REST 只是风格，不是技术规范或标准，所以有些实现 REST 的细节没有明确的定义，这对实践而言，不可避免会产生某些误解。比如在创建和更新某个资源的时候，开发者比较迷茫的是何时该用 HTTP 的 PUT 方法，何时该使用 POST 方法。为了解决这一问题，我们首先应该知道 PUT 方法的特性。PUT 方法是幂等的，即多次插入或者更新同一份数据，在服务器端对资源状态所产生的改变是相同的。PUT 方法不是安全的，有写动作的 HTTP 方法都不是安全的。我们知道，由于使用同一份数据向服务器请求更新某一资源，得到的结果应该总是相同的，因此对于更新操作，使用 PUT 是没有疑问的。可能读者会想到最后更新时间字段每次提交会不同，但那已经不是同一份数据了。

2. 添加资源

创建操作通常每次得到的结果是不同的，因为服务器端的业务层逻辑通常要求数据的主键字段要么来自于业务平台自增一个逻辑值，要么来自于数据库的主键自增。因此，相同的数据每一次提交到服务器端，都会为数据添加一个新的主键值，也就是创建一个主键值不同的新资源（如果没有业务或者外键冲突）。所以，创建操作通常应当设计为 POST 方法的 API。唯有一种场景应当使用 PUT 方法来设计 API，即客户端在发起创建请求时，在同一份数据中总可以提供唯一的主键值，服务器不会对其进行修改，这样的创建请求确保了幂等

性，不应再使用 POST 方法。JAX-RS2 定义了 @PUT 注解来定义相关资源方法，示例代码如下。

```
@Path("book")
public interface BookResource {
    // 关注点 1: PUT 方法
    @PUT
    // 关注点 2: 资源方法定义了 Produces 注解和 Consumes 注解
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes(MediaType.APPLICATION_XML)
    public String newBook(Book book);
}

public class PutTest extends JerseyTest {
    public static AtomicLong clientBookSequence = new AtomicLong();
    @Test
    public void testNew() {
        final Book newBook = new Book(clientBookSequence.incrementAndGet(),
            "book-" + System.nanoTime());
        MediaType contentTypeMediaType = MediaType.APPLICATION_XML_TYPE;
        MediaType acceptMediaType = MediaType.TEXT_PLAIN_TYPE;
        final Entity<Book> bookEntity = Entity.entity(newBook,
            contentTypeMediaType);
        final String lastUpdate = target("book").request(acceptMediaType)
            .put(bookEntity, String.class);
        // 关注点 3: 资源方法定义了 Produces 注解和 Consumes 注解
        Assert.assertNotNull(lastUpdate);
        LOGGER.debug(lastUpdate);
    }
}
```

在这段代码中，资源接口 BookResource 使用 @PUT 注解定义了 newBook() 方法，即该方法用于处理相对资源路径为 "book" 的 PUT 请求，见关注点 1。单元测试类 PutTest 对其功能性进行了验证，对 lastUpdate 使用非空断言，lastUpdate 是更新方法 newBook() 的返回实体的值，代表最后更新时间戳，见关注点 3。我们注意到，newBook() 方法上，同时定义了 @Produces(MediaType.TEXT_PLAIN) 注解和 @Consumes(MediaType.APPLICATION_XML) 注解，见关注点 2，下面我们来介绍一下与关注点 2 相关的媒体类型知识。

3. 媒体类型

PUT 方法执行写操作的非安全的 HTTP 方法，需要考虑请求实体媒体类型和响应实体媒体类型。请求实体媒体类型使用 HTTP 头的 Content Type 定义，响应实体媒体类型使用 HTTP 头的 Accept 定义。

在服务器端，@Consumes(MediaType.APPLICATION_XML) 定义了服务器端要消费的媒体类型，即消费客户端请求实体的媒体类型。@Produces(MediaType.TEXT_PLAIN) 定义了服务器端生产的媒体类型，即服务器产生的响应实体的媒体类型。客户端在提交非安全性 HTTP 请求方法前，在 Entity 类的实例中，定义该 Entity 实例的媒体类型，即客户端请求实

体的媒体类型。request 方法用于定义可接受的 HTTP 方法的返回媒体类型，即服务器的响应实体的媒体类型。

测试资源方法 newBook(), 将得到如下所示的请求头信息，从中可以看到请求媒体类型。

```
public final static String TEXT_PLAIN = "text/plain";
public final static String APPLICATION_XML = "application/xml";
public final static MediaType TEXT_PLAIN_TYPE = new MediaType("text", "plain");
public final static MediaType APPLICATION_XML_TYPE = new MediaType("application", "xml");

1 > PUT http://localhost:9998/book
1 > Accept: text/plain
1 > Content-Type: application/xml
```

在这段代码中，javax.ws.rs.core.MediaType 类是 JAX-RS2 提供的媒体类型定义类，其中定义了包括示例中使用的 MediaType.TEXT_PLAIN，其值为 "text/plain"。在 MediaType 类中，对应的响应实体媒体类型定义为 Accept: text/plain；MediaType.APPLICATION_XML 值为 "application/xml"，对应的请求实体媒体类型定义为 Content-Type: application/xml。

2.1.3 DELETE 方法

DELETE 方法是幂等的，即多次删除同一份数据（通常请求中传递的参数是数据的主键值），在服务器端产生的改变是相同的。JAX-RS2 定义了 @DELETE 注解来定义相关资源方法。下面来看看具体示例。

执行删除的资源方法，其返回值可以定义为 void，即该方法没有返回值。之所以在删除资源的场景中可以采用这样的方式定义，是因为删除的前提是对该资源信息已经充分了解，没有必要再将其从服务器上传递回来，示例代码如下。

```
@Path("book")
public interface BookResource {
    @DELETE
    public void delete(@QueryParam("bookId") final long bookId);
}
```

在这段代码中，无返回值的资源方法 delete() 返回的响应实体为空，HTTP 状态码为 204。该定义可以参考 Jersey 的源代码中的 Response 类，示例代码如下。

```
package javax.ws.rs.core;
public abstract class Response {
    public interface StatusType {
        public enum Status implements StatusType {
            NO_CONTENT(204, "No Content"),

```

接下来是删除资源方法的单元测试，示例代码如下。

```
public class DeleteTest extends JerseyTest {
```

```

@Test
public void testGet() {
    final Response response = target("book").queryParam("bookId", "9527")
        .request().delete();
    int status = response.getStatus();
    LOGGER.debug(status);
    Assert.assertEquals(Response.Status.NO_CONTENT.getStatusCode(), status);
}
}

```

在这段代码中，对 REST 请求的测试断言不是针对删除资源的实体，而是响应中 HTTP 状态码。也就是说，删除资源方法的返回值类型可以定义为 void，业务逻辑更关注删除操作的结果状态。

2.1.4 POST 方法

POST 方法是一种写操作的 HTTP 请求。RPC 的所有写操作均使用 POST 方法，而 REST 只使用 HTTP 的 POST 方法添加资源。

1. 既不幂等也不安全

定义为 POST 的 REST 接口用于写数据，POST 方法的特性是既不幂等也不安全。由于请求会改变服务器端资源的状态，因此它是不是安全的；由于每次请求对服务器端资源状态的改变并不是相同的，因此它不是幂等的。

2. 两种分类

REST 中使用的 POST 可以称之为 POST (a)，即用于创建、添加资源的 HTTP 方法。这是相对于 RPC 式的 Web 服务中对 POST 的使用而言的。

在 RPC 中使用的 POST 可以称之为 POST (p)，即通过重载的 POST 用于处理某种操作。服务器接收 POST (p) 的请求后，不是直接处理 POST 请求，由于真正的方法信息位于信封头或实体主体里，因此需要先解析出执行方法。

JAX-RS2 定义了 @POST 注解来定义相关资源方法。示例代码如下。

```

@Path("book")
public interface BookResource {
    // 关注点 1: POST 方法
    @POST
    @Produces(MediaType.APPLICATION_XML)
    @Consumes(MediaType.APPLICATION_XML)
    public Book createBook(Book book);

    public class PostTest extends JerseyTest {
        @Test
        public void testCreate() {
            final Book newBook = new Book("book-" + System.nanoTime());
            MediaType contentTypeMediaType = MediaType.APPLICATION_XML_TYPE;

```



```

MediaType acceptMediaType = MediaType.APPLICATION_XML_TYPE;
final Entity<Book> bookEntity = Entity.entity(newBook, contentTypeMediaType);
final Book book =target("book").request(acceptMediaType).post(bookEntity,
    Book.class);
// 关注点 2: 测试 POST 方法的断言
Assert.assertNotNull(book.getBookId());
LOGGER.debug("Server Id="+book.getBookId());
}
}

```

在这段代码中，资源接口 `BookResource` 定义了 `createBook()` 方法，该方法使用 `@POST` 注解，表示该方法处理 "book" 路径下的 POST 请求，见关注点 1。在测试方法 `testCreate()` 中，关注请求结果实体的主键是否为空。这是因为在 POST 请求提交的添加资源操作中，主键的设置是在服务器端完成的，因此客户端成功请求添加资源后，应关注服务器端返回的实体结果是否有主键信息，见关注点 2。

到此，我们完成了对 HTTP 的基本方法的讲述。除了 HTTP 协议定义的标准方法，还存在来自其他协议中的 HTTP 方法。接下来，我们一起探讨这些方法对 REST 服务的影响。

2.1.5 WebDAV 扩展方法

WebDAV (Web-based Distributed Authoring and Versioning，基于 Web 的分布式创作与版本控制) 是 IETF 的 RFC4918 规范 (RFC2518 规范的替代规范地址是 <http://tools.ietf.org/html/rfc4918>)，是对 HTTP1.1 协议的一组扩展，该协议允许用户以协作方式编辑和管理远程 Web 服务器上的文件。WebDAV 在 HTTP 方法的基础上，增加了如下方法 (详见 RFC4918 第 9 章)。

- ❑ **PROPFIND 方法**：用于从 Web 资源中查询存储为 XML 格式的属性数据，或者重载为从一个远程系统中查询目录结构的数据。
- ❑ **PROPPATCH 方法**：用于原子地更改和删除一个资源的多个属性。
- ❑ **MKCOL 方法**：用于创建目录。
- ❑ **COPY 方法**：用于将资源从一个 URI 资源地址复制到另一个 URI 资源地址。
- ❑ **MOVE 方法**：用于将资源从一个 URI 资源地址移动到另一个 URI 资源地址。
- ❑ **LOCK 方法**：用于锁定一个资源。WebDAV 支持共享锁和独占锁。
- ❑ **UNLOCK 方法**：用于解锁一个资源。

虽然 WebDAV 对 HTTP 方法做出了功能性扩展，使之提供更强大服务，但是从 ROA 角度讲，因为 WebDAV 在 HTTP 标准方法的基础上增加了特殊的方法名称，WebDAV 破坏了统一接口的原则。因此，对是否应该在 REST 式的 Web 服务中支持 WebDAV，业内的观点并不一致。

笔者的观点是如果遵从 ROA，那么就不使用 HTTP 标准方法之外的方法。如果业务需求确实超出了标准方法所及，那么可以使用如下注解实现对 WebDAV 的支持。JAX-RS2 规范没有阐述对 WebDAV 提供支持的文字，但是 JAX-RS2 定义了 `@HttpMethod` 注解来定义相关的资源方法。在 Jersey 应用中，可以使用 `@HttpMethod` 注解定义 HTTP 标准方法之外的方法名称来支持 WebDAV，示例代码如下。

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(value = "MOVE")
@Documented
public @interface MOVE {
}
```

这段代码是对 `@MOVE` 注解的定义，使用 `@HttpMethod` 注解定义了名为 MOVE 的 HTTP 扩展方法。有了扩展方法注解，我们就可以在资源类中定义新的方法来支持扩展方法的请求了，示例代码如下。

```
@Path("book")
public interface BookResource {
    @MOVE
    public boolean moveBooks(Books books);
}
```

在这段代码中，资源类 `BookResource` 定义了 `moveBooks` 方法，该方法使用 `@MOVE` 注解定义，表示用于处理 "book" 路径下的 MOVE 请求。下面我们来看看相关的测试代码。

```
public class HttpMethodTest extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig resourceConfig = new ResourceConfig(EBookResourceImpl.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig clientConfig) {
        // 关注点 1: 定义 Grizzly 连接器
        clientConfig.connectorProvider(new GrizzlyConnectorProvider());
        super.configureClient(clientConfig);
    }
    @Test
    public void testWebDav() {
        // 关注点 2: HTTP MOVE 请求
        final Response response = target("book").request().method("MOVE");
        Boolean result = response.readEntity(Boolean.class);
        // 关注点 3: Move 方法测试断言
        Assert.assertEquals(Boolean.TRUE, result);
    }
}
```

在这段代码中，测试方法 `testWebDav()` 在请求中定义了 MOVE 请求，见关注点 2；断

言是针对 MOVE 方法的返回值，见关注点 3；可以看出，使用 Jersey 实现对 WebDav 的支持并不困难。

需要注意的是，Jersey 默认的连接器只支持 HTTP 标准方法，因此要使用 HTTP 的扩展方法就不能直接使用默认的连接器，这里使用了 Grizzly 连接器。对应的代码行是：`clientConfig.connectorProvider(new GrizzlyConnectorProvider())`，即为客户端配置实例提供 Grizzly 连接器，见关注点 1。这行代码是 Jersey2.5+ 后的写法，Jersey2.5 之前的写法是 `clientConfig.connector(new GrizzlyConnector(clientConfig))`。从中可以看出，Jersey 在不断优化中，包括 API。这一好处是活跃的社区为用户带来越来越便捷、高效的使用体验，缺点是破坏了向下兼容性。

到此，我们全面掌握了 HTTP 方法在 REST 统一接口定义中的作用和实现。明白了 REST 接口该使用什么样的请求方法非常重要，这决定了其性质。但是这还不够，一个接口如何被请求唯一定位还需要深入掌握 REST 的资源定位。接下来一节将详述资源定位的细节。

2.2 资源定位

REST 使用 URI 实现资源定位，从这个角度上讲，对外提供 REST 式的 Web 服务的接口就是公布一系列的 URI 及其参数，这使得 REST 的实践过程简单到了极致。但是 URI 形式上的简单并不意味着我们可以将 URI 的定义信手拈来，正所谓“没有规矩，不成方圆”。

在设计 REST 式的 Web 服务过程中，资源地址的设计是非常严谨的，如果设计不得体，不仅 REST 接口的风格无法统一，使系统的扩展性和易用性降低，也很难实现资源准确地被定位。

资源地址的设计过程是面向资源的，资源名称应是准确描述该资源的名词，资源地址应具有直观的描述性。比如一个班级的资源地址可以是：学校 / 学院 / 年级 / 班级。值得注意的是一个 URI 资源地址唯一对应一个资源，但是一个资源可以拥有多个 URI 资源地址。比如 Jersey 最新版本的文档地址和 Jersey2.7 版本的文档地址指向同一个资源（本书写作时）。

阅读指南

本节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：`com.example.annotation.param`。

2.2.1 资源地址设计

资源地址的设计对整个 REST 式的 Web 服务至关重要，涉及系统的可用性、可维护性和可扩展性等诸多方面的表现。因此，本节关注如何对资源地址进行设计。

1. 资源路径概览

资源地址的路径变量是用来表达逻辑上的层次结构的，资源和子资源的形式是自左至右、斜杠分割的名词。它们的关系可以是整体到局部，比如学校到班级，城市到乡镇；也可以是从一般到具体，比如一个生物的“门、纲、目、科、属、种”的资源路径。资源地址具体可以分为 5 个部分，以 `scheme://host:port/path?queryString` 为例，如表 2-1 所示。

表 2-1 资源地址路径分解

| 元素 | 描 述 |
|-------------|---|
| scheme | 协议名称，通常是 HTTP 和 HTTPS |
| host | (DNS) 主机名称或者 IP 地址 |
| port | 服务端口 |
| path | 资源地址，使用“/”符号来分隔逻辑上的层次结构 |
| ? | 用来分隔资源地址和查询字符串符号 |
| queryString | 查询字符串，方法作用域信息 使用“&”符号来分隔查询条件 使用逗号分隔有次序的作用域信息 使用分号分隔无次序的作用域信息 |

一个典型的 URI 如表 2-1 所示，包括协议名称、主机名称、服务端口、资源地址和查询字符串等 5 个部分。其中资源地址部分，根据具体部署的不同或有差别，如图 2-2 所示。

`http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1`

图 2-2 资源地址示例

图 2-2 中，通常使用 ContextPath、ServletPath 和 PathInfo 来细分资源地址。ContextPath 是上下文名称，通常和部署服务器的配置或者 REST 服务的 web.xml 配置有关。ServletPath 是 Servlet 的名称，与 REST 服务中定义的 @ApplicationPath 注解或者 web.xml 的配置有关。JAX-RS2 定义了 @Path 注解来定义资源地址。PathInfo 是资源路径信息，与资源类、子类以及类中的方法定义的 @Path 注解有关。

现在我们对资源地址的层次结构有了认识，此时需要思考一个问题：资源地址是否可以唯一定位一个资源？

答案是否定的。资源地址相同，但 HTTP 方法不同的两个方法是两个不同的 REST 接口。HTTP 方法和资源地址结合在一起才可以完成对资源的定位。细心的读者也许已经从 3.1 节的示例中看出端倪。示例中，GET 方法用于读取 / 检索、查询 / 过滤一个资源，PUT 方法用于修改 / 更新资源、创建客户端维护主键信息的资源，DELETE 方法用于删除资源，POST 方法用于创建资源。但这些方法的资源地址是相同的，都是 "book"。

当上述的标准 HTTP 方法无法满足业务需求时，比如对于图书资源，除了基本的 CRUD 之外，若需要公布像借阅、折旧、电子版下载等实际生活中的更新操作的接口时，单单公布一个 PUT 方法就不够用了。这些操作是动词性的，无法简单地使用一个 book 名词定位。在路径变量难以准确描述的情况下，一种方案是可以考虑使用动词作为查询参数；另一种方案是可以在 REST 设计过程中引入 RPC 风格的 POST 方法，辅助完成复杂业务的接口设计，这就是 REST 和 RPC 混合型的 Web 服务了。

2. 资源地址和作用域

在路径变量里可以使用标点符号以辅助增强逻辑清晰性。这些辅助符号用在表 2-2 中的查询字符串，作为资源地址的查询变量，用来表达算法的输入，实现对方法的作用域的约束。下面来逐一讲述这些对资源地址设计至关重要的符号。

(1) 问号 (?) 是用来分隔资源地址和查询字符串的，与符号 (&) 是用来分隔查询条件的参数的。示例代码如下。

```
GET /books?start=0&size=10
```

这行代码中的作用是查询图书列表，开始行参数为 0，条目参数为 10，即从第 0 行开始取 10 条并返回该图书列表。

(2) 逗号 (,) 是用来分隔有次序的作用域信息。需要注意的是逗号分隔的逻辑上的顺序信息，这种顺序可以是约定俗成的，比如先写经度后写纬度；也可以是系统约定的，比如月、日、年的顺序等。举例来说，按时间区间查询图书，日期信息在资源地址中是采用月、年顺序，示例如下。

```
GET /books/01,2002-12,2014
```

这行代码中的作用是查询 2002 年 1 月到 2014 年 12 月这个时间段 (出版) 的图书。这个例子中还使用了连字符 (-)，有时候也可以使用下横线 (_) 来做逻辑上的辅助分隔。

(3) 分号 (;) 是用来分隔无次序的作用域信息。通常这些信息是逻辑上并列存在的，比如并列的查询条件，示例如下所示。

```
GET /books/restful;program=java;type=web
```

这行代码中的作用是查询满足图书内容为 restful 的、使用的编程语言是 Java 的、讲

述的类型是 Web 的图书列表。这样的逻辑没有顺序，互换顺序的查询条件不会影响资源的表述。

基于上述理论，这里抛砖引玉，列出常用的资源地址设计示例如表 2-2 所示。

表 2-2 资源地址设计

| 功能 | 资源地址 |
|---------|--|
| 添加 / 创建 | POST /books |
| | PUT /books/{id} |
| 删除 | DELETE /books/{id} |
| 修改 / 更新 | PUT /books/{id} |
| 查询全部 | GET /books HTTP1.1 |
| 主键查询 | GET /books/{id} HTTP1.1 |
| | GET /books?id=12345678 |
| 分页作用域查询 | GET /books?start=0&size=10 |
| | GET /books/01,2002-12,2014 |
| | GET /books/restful;program=java;type=web |
| | GET /books?limit=100&sort=bookname |

如果读者可以轻松领会表 2-2 列出的这些典型的 REST 接口和资源定位的设计，就可以放手实现了，否则建议回顾本节内容。接下来，我们完成从设计到实现的跨越，看看 JAX-RS2 标准是如何通过注解来支持资源定位的，并使用 Jersey 完成上述设计的实践。

2.2.2 @QueryParam 注解

查询条件决定了方法的作用域，查询参数组成了查询条件。JAX-RS2 定义了 @QueryParam 注解来定义查询参数，本节使用 @QueryParam 演示 3 个 REST 查询接口的实现示例如表 2-3 所示。

表 2-3 @QueryParam 示例列表

| 接口描述 | 资源地址 |
|-------------|---|
| 分页查询列表数据 | /query-resource/yijings?start=24&size=10 |
| 排序并分页查询列表数据 | /query-resource/sorted-yijings?limit=5&sort=pronounce |
| 查询单项数据 | /query-resource/yijing?id=8 |

1. 分页查询

分页查询是使用 @QueryParam 解析参数的基本示例，实现代码如下所示。

```
public Yijings getByPaging(@QueryParam("start")final int start,
    @QueryParam("size")final int size){// 关注点 1: 资源方法入参
    ...
    int listSize = globalList.size();
```

```

        final int max = size > listSize ? listSize : size;
// 关注点 2: 分页迭代逻辑
        for(int i = 0, index = start; i < max; i++) {
            final Yijing yijing = globalList.get(index + i);
// 关注点 3: 添加 Link 以保证 REST 的连通性
            final URI location = ub.clone().queryParam("id", yijing.getSequence()).build();
            final Link link =
new Link("detail", location.toASCIIString(), MediaType.APPLICATION_XML);
            links.add(link);
            yijings.add(yijing);
        }
        result.setLinks(links);
        result.setGuas(yijings);
        return result;
    }

```

在这段代码中, `getByPaging()` 方法的输入参数包含了 2 个使用 `@QueryParam` 注解定义的查询参数, 分别是起始条目参数 "start" 和条目数量参数 "size", 参数的类型是整型, 见关注点 1。在查询的迭代中使用这两个参数获取图书列表, 见关注点 2。在迭代中, 每个图书资源条目的 URI 都存储在返回值中, 以保证资源的联通性, 见关注点 3。该 URI 被封装到 `Link` 实例中, 在单项查询时使用。

另外, 参数的定义使用了 `final`, 符合 Checkstyle 的编程风格, 即输入参数只作为逻辑算法的依据使用, 其本身不会在这过程中被修改。也许这种不变的变量对提高执行效率并没有多少影响, 但跬步积千里、蚁穴溃长堤。推荐 Java 开发者在 REST 开发中引入 SonarQube 平台或者单纯使用 Checkstyle 工具对静态代码进行质量检测, 以帮助我们改进代码的质量。

2. 排序查询

排序查询是在解析参数的基础上, 额外处理结果集顺序的示例, 代码如下。

```

public Yijings getByOrder(@QueryParam("limit") final int limit,
    @QueryParam("sort") final String sortName) { // 关注点 1: 资源方法入参
    ...
    Collections.sort(list, new Comparator<Yijing>() {
        @Override
// 关注点 2: 排序中的比较算法
        public int compare(final Yijing o1, final Yijing o2) {
            switch (sortName) {
                case "sequence":
                    return o1.getSequence().compareTo(o2.getSequence());
                case "name":
                    return o1.getName().compareTo(o2.getName());
                case "pronounce":
                    return o1.getPronounce().compareTo(o2.getPronounce());
            }
            return 0;
        }
    });
}

```

在这段代码中, limit 参数的用途同分页查询示例, 而 sortName 参数则用于排序, 见关注点 1; 排序接口需要额外解析 sortName 传递的排序字段, 并将其作为数据库查询语句中的排序参数使用。这里实现了 Comparator 接口的 compare() 方法来完成根据不同字段对集合的排序, 见关注点 2。

3. 单项查询

客户端在获得结果集的基础上, 根据表述中链接信息, 向服务器发起单项查询的示例, 代码示例如下所示。

```
public Yijing getByQuery(@QueryParam("id") final int seqId) {
    return ParamCache.find("" + seqId);
}
```

在这段代码中, 使用 @QueryParam 定义了 "id" 参数, 该参数来自分页查询中返回的 URI 信息。

注解 QueryParam 可以和注解 DefaultValue 一起使用。注解 DefaultValue 的作用是预置一个默认值, 当请求中不包含此参数时使用, 示例如下。

```
@DefaultValue("100") @QueryParam("size") final Integer pageSize
```

这句话的意思是当请求中不包含分页参数 pageSize 时, 分页参数 pageSize 的默认值为 100。

2.2.3 @PathParam 注解

JAX-RS2 定义了 @PathParam 注解来定义路径参数——每个参数对应一个子资源, 本节使用 @PathParam 完成如表 2-4 所示的 REST 查询接口。

表 2-4 @PathParam 示例列表

| 接口描述 | 资源地址 |
|-------------|---|
| 基本路径参数 | /path-resource/Eric |
| 结合查询参数 | /path-resource/Eric?hometown=Buenos Aires |
| 带有标点符号的资源路径 | /path-resource/199-1999 /path-resource/01,2012-12,2014 |
| 子资源变长的资源路径 | /path-resource/Asia/China/northeast/liaoning/shenyang/huangu /path-resource/q/restful;program=java;type=web /path-resource/q2/restful;program=java;type=web |

1. @Path 注解

JAX-RS2 定义了 @Path 注解来定义资源路径, @Path 接收一个 value 参数来解析资源路径地址。该参数除了前面示例中的 books 这种静态定义的方式外, 也可以使用动态

变量的方式，其格式为：{ 参数名称:正则表达式 }。这个接口的功能和查询参数实现的 /query-resource/yijings?start=24&size=10 相似，也是用于分页查询，其资源地址形如：/path-resource/199-1999，参考示例如下。

```
@GET
@Path("{from:\\d+}-{to:\\d+}")
public String getByCondition(@PathParam("from") final Integer from,
    @PathParam("to") final Integer to) {
    ...
}
```

在这段代码中，使用 @PathParam 注解定义的两个参数 from 和 to 用以定义查询区间，正则表达式部分是 \\d+，表示数字。两个参数中间的连接符 (-) 是路径的格式信息。稍显复杂的例子是：/path-resource/01,2012-12,2014，引入了逗号 (,) 作为有顺序的日期分隔符号，那么对应的正则表达式为：@Path("{beginMonth:\\d+},{beginYear:\\d+}-{endMonth:\\d+},{endYear:\\d+}")

2. 正则表达式

正则表达式的讲述超出了本书范围，这里只简述示例中用到的正则表达式。刚刚的例子中的 \\d+，代表参数应为数字并且至少出现一次。第一个反斜杠是 Java 中的转义字符，第二个反斜杠是正则表达式的起始，加号 (+) 是至少出现一次的意思，星号 (*) 则代表出现至少零次，句号 (.) 是匹配任何字符，d 是匹配数字，w 是匹配数字和字母。我们有的放矢，示例中使用的正则表达式如表 2-5 所示，读者掌握所列的路径含义即可，我们的目的是学习 REST API 设计，而非正则本身。

表 2-5 正则表达式示例

| 正则表达式 | 含 义 |
|---------------------------|---|
| [a-zA-Z][a-zA-Z_0-9]* | 以字母开头，后面是零到多个“字母_数字”格式的字符组合 |
| {region:.+}/{district:w+} | region 变量至少包含一个任意字符。 district 变量至少包含一个为数字或者字母的字符 |

3. 路径配查询

查询参数和路径参数在一个接口中配合使用，可以更便捷地完成资源定位，这很像战场上的多兵种协同作战。前述的图书资源的复杂设计就需要两者结合来完成，示例代码如下。

```
@Path("{user: [a-zA-Z][a-zA-Z_0-9]*}")
@Produces(MediaType.TEXT_PLAIN)
public String getUserInfo(@PathParam("user") final String user,
    @DefaultValue("Shen Yang")@QueryParam("hometown") final String hometown) {
    return user + ":" + hometown;
}
```

在这段代码中，路径参数 `user` 中使用了通配符，方法参数中同时使用 `@PathParam` 注解和 `@QueryParam`，定义了 `user` 和 `hometown` 两个参数。以资源地址：`/path-resource/Eric?hometown=Buenos Aires` 为例，REST 容器会将该请求匹配到 `getUserInfo()` 方法，其中 `Eric` 是路径变量 `user` 的值，`Buenos Aires` 作为查询变量 `hometown` 的值。

4. 路径区间

路径区间 (`PathSegment`) 是对资源地址更灵活的支持，使资源类的一个方法可以支持更广泛的资源地址的请求。我们从下面定义的资源地址列表来走近 `PathSegment`。

```
/path-resource/Asia/China/northeast/liaoning/shenyang/huangu
/path-resource/China/northeast/liaoning/shenyang/tiexi
/path-resource/China/shenyang/huangu
```

如上所示的资源地址中含有固定子资源 (`shenyang`) 和动态子资源两部分。对于动态匹配变长的子资源资源地址，`PathSegment` 类型的参数结合正则表达式将大显身手，示例代码如下。

```
@GET
@Path("/{region:.+}/shenyang/{district:\\w+}")
public String getByAddress(@PathParam("region") final List<PathSegment> region,
    @PathParam("district") final String district) {
    final StringBuilder result = new StringBuilder();
    for (final PathSegment pathSegment : region) {
        result.append(pathSegment.getPath()).append("-");
    }
    result.append("shenyang-" + district);
    ...
}
```

在这段代码中，`getByAddress()` 方法用来匹配表的这些资源地址。该方法的 `region` 变量是 `PathSegment` 类型的数组，以匹配至少出现一个字符的正则表达式 (`+`)。 `PathSegment` 如其名字所示，是路径的片段，是子资源的集合。遍历 `PathSegment` 集合，对于每一个 `PathSegment` 实例，可以通过调用其 `getPath()` 方法获取子资源名称。

对于查询参数动态给定的场景，可以定义 `PathSegment` 作为参数类型，通过 `getMatrixParameters()` 方法获取 `MultivaluedMap` 类型的查询参数信息，即可将参数条件作为一个整体解析，示例代码如下。

```
@Path("/q/{condition}")
public String getByCondition3(@PathParam("condition") final PathSegment
    condition) {
    ...
    final MultivaluedMap<String, String> matrixParameters = condition.
        getMatrixParameters();
    final Iterator<Entry<String, List<String>>>
        iterator = matrixParameters.entrySet().iterator();
```



```

while (iterator.hasNext()) {
    final Entry<String, List<String>> entry = iterator.next();
    conds.append(entry.getKey()).append("=");
    conds.append(entry.getValue()).append(" ");
}
return conds.toString();
}

```

在这段代码中，`getByCondition3()` 方法只有一个 `PathSegment` 类型的参数 `condition`，该参数包含了查询条件中携带的全部参数列表。举例来说，资源地址为 `path-resource/q/restful;program=java;type=web` 的请求可以匹配到 `getByCondition3()` 方法，其中，`MultivaluedMap` 类型的实例 `matrixParameters` 的值为 `[program=[java], type=[web]]`。

5. @MatrixParam 注解

上例中，通过编程方式，调用 `PathSegment` 类的 `getMatrixParameters()` 方法来获取查询参数信息。还有一种方式是通过 `@MatrixParam` 注解来逐一定义参数，即通过声明方式来获取，示例代码如下。

```

@Path("q2/{condition}")
public String getByCondition4(@PathParam("condition")
    final PathSegment condition, @MatrixParam("program") final String program,
    @MatrixParam("type") final String type) {
    return condition.getPath() + " program=[" + program + "] type=[" + type + "];"
}

```

在这段代码中，使用 `@MatrixParam` 注解分别定义了 "program" 和 "type" 两个参数。与上例相比，这段代码更能清晰地表达可接收的参数名称和类型，缺点是缺乏对请求资源地址更灵活的支持。

2.2.4 @FormParam 注解

JAX-RS2 定义了 `@FormParam` 注解来定义表单参数，相应的 REST 方法用以处理请求实体媒体类型为 `Content-Type: application/x-www-form-urlencoded` 的请求，示例代码如下。

```

@Path("form-resource")
public class FormResource {
    @POST
    public String newPassword(
        @DefaultValue("feuyeux") @FormParam(FormResource.USER) final String user,
        @Encoded @FormParam(FormResource.PW) final String password,
        @Encoded @FormParam(FormResource.NPW) final String newPassword,
        @FormParam(FormResource.VNPW) final String verification) {

```

在这段代码中，`newPassword()` 方法是 `@FormParam` 注解定义了 `user` 等 4 个参数，这些参数是容器从请求中获取并匹配的。相关的客户端测试如图 2-3 所示。

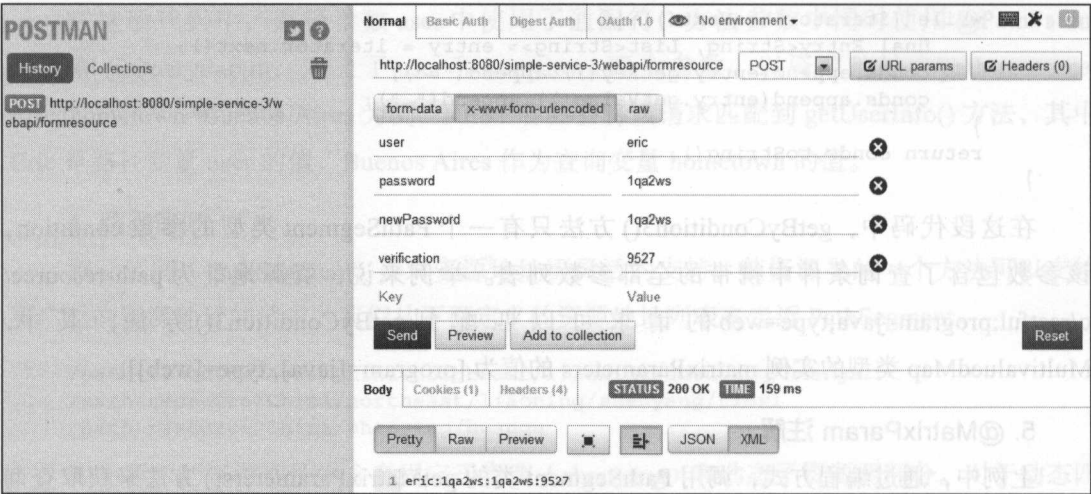


图 2-3 表单示例

图 2-3 所示的客户端工具是 POSTMAN（详见 2.6 节），使用 POSTMAN 定义的基本表单信息与 newPassword() 方法一致。

newPassword() 方法的测试代码片段，示例代码如下。

```
@Test
public void testPost2() {
    final Form form = new Form();
    form.param(FormResource.USER, "feuyeux");
    form.param(FormResource.PW, "北京");
    form.param(FormResource.NPW, "上海");
    form.param(FormResource.VNPW, "上海");
    final String result = target("form-resource").request().
    post(Entity.entity(form, MediaType.APPLICATION_FORM_URLENCODED_TYPE), String.class);
    FormTest.LOGGER.debug(result);
    Assert.assertEquals("encoded should let it to disable decoding",
    "feuyeux:%E5%8C%97%E4%BA%AC:%E4%B8%8A%E6%B5%B7:上海", result);
}
```

在这段代码中，Form 类实例是请求实体，请求实体的类型为 MediaType.APPLICATION_FORM_URLENCODED_TYPE，即 application/x-www-form-urlencoded。这里还需要注意的是 @Encoded 注解和 @DefaultValue 注解的使用。

JAX-RS2 定义了 @Encoded 注解用以标识禁用自动解码。示例的测试结果中 “%E4%B8%8A%E6%B5%B7” 是 newPassword() 方法的参数值“上海”的编码值，当对 newPassword 使用 @Encoded 注解，REST 方法得到的参数值就不会被解码，如果将其直接返回，那么客户端得到的值就会是处于编码状态的字符串。

JAX-RS2 定义了 @DefaultValue 注解，用以为客户端没有为其提供值的参数提供默认

值。本例的 user 参数的默认值为 feuyeux。

2.2.5 @BeanParam 注解

JAX-RS2 定义了 @BeanParam 注解用于自定义参数组合，使 REST 方法可以使用简洁的参数形式完成复杂的接口设计。@BeanParam 注解的使用示例如下所示。

```
@GET
@Path("{region:.+}/shenyang/{district:\\w+}")
// 关注点 1: 资源方法入参
public String getByAddress(@BeanParam Jaxrs2GuideParam param) {
    // 关注点 2: 参数组合
    public class Jaxrs2GuideParam {
        @HeaderParam("accept")
        private String acceptParam;
        @PathParam("region")
        private String regionParam;
        @PathParam("district")
        private String districtParam;
        @QueryParam("station")
        private String stationParam;
        @QueryParam("vehicle")
        private String vehicleParam;

        public void testBeanParam() {
            ...
            final WebTarget queryTarget = target(path).path("China").path("northeast")
                .path("shenyang").path("tiexi")
                .queryParam("station", "Workers Village").queryParam("vehicle", "bus");
            result = queryTarget.request().get().readEntity(String.class);
            // 关注点 3: 查询结果断言
            Assert.assertEquals("China/northeast:tiexi:Workers Village:bus", result);
        }

        // 关注点 4: 复杂的查询请求
        http://localhost:9998/ctx-resource/China/shenyang/tiexi?station=Workers+Villag
        e&vehicle=bus
    }
}
```

在这段代码中，getByAddress() 方法只用了一个使用 @BeanParam 注解定义的 Jaxrs2GuideParam 类型的参数，见关注点 1；Jaxrs2GuideParam 类定义了一系列 REST 方法会用到的参数类型，包括示例中使用的查询参数 "station" 和路径参数 "region" 等，从而使得 getByAddress() 方法可以匹配更为复杂的资源路径，见关注点 2；在变长子资源的例子基础上，增加了查询条件，但测试方法 testBeanParam() 发起的请求的资源地址见关注点 4；可以看出这是一个较为复杂的查询请求。其中路径部分包括 China/shenyang/tiexi，查询条件包括 station=Workers+Village 和 vehicle=bus。这些条件均在 Jaxrs2GuideParam 类中可以匹配，因此从关注点 3 的测试断言中可以看出，该请求响应的预期结果是 "China/northeast:tiexi:Workers Village:bus"。

2.2.6 @CookieParam 注解

JAX-RS2 定义了 @CookieParam 注解用以匹配 Cookie 中的键值对信息，示例如下。

```
@GET
public String getHeaderParams(@CookieParam("longitude") final String longitude,
    @CookieParam("latitude") final String latitude,
    @CookieParam("population") final double population,
    @CookieParam("area") final int area) { // 关注点 1: 资源方法入参
    return longitude + "," + latitude + " population=" + population + ",area=" + area;
}

@Test
public void testContexts() {
    final Builder request = target(path).request();
    request.cookie("longitude", "123.38");
    request.cookie("latitude", "41.8");
    request.cookie("population", "822.8");
    request.cookie("area", "12948");
    result = request.get().readEntity(String.class);
    // 关注点 2: 测试结果断言
    Assert.assertEquals("123.38,41.8 population=822.8,area=12948", result);
}
```

在这段代码中，getHeaderParams() 方法包含 4 个使用 @CookieParam 注解定义的参数，用于匹配 Cookie 的字段，见关注点 1；在测试方法 testContexts 中，客户端 Builder 实例填充了相应的 cookie 键值对信息，其断言是对 cookie 字段值的验证，见关注点 2。

2.2.7 @Context 注解

JAX-RS2 定义了 @Context 注解来解析上下文参数，JAX-RS2 中有多种元素可以通过 @Context 注解作为上下文参数使用，示例代码如下。

```
public String getByAddress(
    @Context final Application application,
    @Context final Request request,
    @Context final javax.ws.rs.ext.Providers provider,
    @Context final UriInfo uriInfo,
    @Context final HttpHeaders headers){
```

在这段代码中，分别定义了 Application、Request、Providers、UriInfo 和 HttpHeaders 等 5 种类型的上下文实例。从这些实例中可以获取请求过程中的重要参数信息，示例代码如下。

```
final MultivaluedMap<String, String> pathMap = uriInfo.getPathParameters();
final MultivaluedMap<String, String> queryMap = uriInfo.getQueryParameters();
final List<PathSegment> segmentList = uriInfo.getPathSegments();
final MultivaluedMap<String, String> headerMap = headers.getRequestHeaders();
```

在这段代码中，UriInfo 类是路径信息的上下文，从中可以获取路径参数集合 getPathParameters() 和查询参数集合 getQueryParameters()。类似地，我们可以从 HttpHeaders 类中获取头信息集合 getRequestHeaders()。这些业务逻辑处理中常用的辅助信息的获取，要通过 @

Context 注解定义方法的参数或者类的字段来实现。

到此，统一接口和资源定位的设计和实现已经讲述完毕。但是，设计 REST 接口还需要在此基础上，掌握请求实体和响应实体的传输格式。接下来让我们看看 Jersey 都支持哪些类型的传输格式。

2.3 传输格式

本节要考虑的就是如何设计表述，即传输过程中数据采用什么样的数据格式。通常，REST 接口会以 XML 和 JSON 作为主要的传输格式，这两种格式数据的处理是本节的重点。那么 Jersey 是否还支持其他的数据格式呢？答案是肯定的，让我们逐一掌握各种类型的实现。

2.3.1 基本类型

Java 的基本类型又叫原生类型，包括 4 种整型（byte、short、int、long）、2 种浮点类型（float、double）、Unicode 编码的字符（char）和布尔类型（boolean）。

阅读指南

本节的前 4 小节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：com.example.response。

Jersey 支持全部的基本类型，还支持与之相关的引用类型。前述示例已经呈现了整型（int）等 Java 的基本类型的参数，本例展示字节数组类型作为请求实体类型、字符串作为响应实体类型的示例，示例代码如下。

```
@POST
@Path("b")
public String postBytes(final byte[] bs) { // 关注点 1：测试方法入参
    for (final byte b : bs) {
        LOGGER.debug(b);
    }
    return "byte[]:" + new String(bs);
}

@Test
public void testBytes() {
    final String message = "TEST STRING";
    final Builder request = target(path).path("b").request();
    final Response response = request.post();
}
```



```
Entity.entity(message, MediaType.TEXT_PLAIN_TYPE), Response.class);
result = response.readEntity(String.class);
// 关注点 2: 测试断言
Assert.assertEquals("byte[]:" + message, result);
}
```

在这段代码中，资源方法 `postBytes()` 的输入参数是 `byte[]` 类型，输出参数是 `String` 类型，见关注点 1；单元测试方法 `testBytes()` 的断言是对字符串 "TEST STRING" 的验证，见关注点 2。

2.3.2 文件类型

Jersey 支持传输 `File` 类型的数据，以方便客户端直接传递 `File` 类实例给服务器端。文件类型的请求，默认使用的媒体类型是 `Content-Type: text/html`，示例代码如下。

```
@POST
@Path("/f")
// 关注点 1: 测试方法入参
public File postFile(final File f) throws FileNotFoundException, IOException {
// 关注点 2: try-with-resources
    try (BufferedReader br = new BufferedReader(new FileReader(f))) {
        String s;
        do {
            s = br.readLine();
            LOGGER.debug(s);
        } while (s != null);
        return f;
    }
}

@Test
public void testFile() throws FileNotFoundException, IOException {
// 关注点 3: 获取文件全路径
    final URL resource = getClass().getClassLoader().getResource("gua.txt");
// 关注点 4: 构建 File 实例
    final String file = resource.getFile();
    final File f = new File(file);
    final Builder request = target(path).path("f").request();
// 关注点 5: 提交 POST 请求
    Entity<File> e = Entity.entity(f, MediaType.TEXT_PLAIN_TYPE);
    final Response response = request.post(e, Response.class);
    File result = response.readEntity(File.class);
    try (BufferedReader br = new BufferedReader(new FileReader(result))) {
        String s;
        do {
            s = br.readLine();// 关注点 6: 逐行读取文件
            LOGGER.debug(s);
        } while (s != null);
    }
}
```

在这段代码中，资源方法 `postFile()` 的输入参数类型和返回值类型都是 `File` 类型，见关注点 1；服务器端对 `File` 实例进行解析，最后将该资源释放，即 `try-with-resources`，见关注

点2；在测试方法 `testFile()` 中，构建了 `File` 类型的 "gua.txt" 文件的实例，见关注点3；作为请求实体提交，见关注点4；并对响应实体进行逐行读取的校验，见关注点5；需要注意的是，由于我们使用的是 Maven 构建的项目，测试文件位于测试目录的 `resources` 目录，其相对路径为 `/simple-service-3/src/test/resources/gua.txt`，获取该文件的语句为 `getClass().getClassLoader().getResource("gua.txt")`，见关注点6。

另外，文件的资源释放使用了 JDK7 的 `try-with-resources` 语法，见关注点2。

2.3.3 InputStream 类型

Jersey 支持 Java 的两大读写模式，即字节流和字符流。本示例展示字节流作为 REST 方法参数，示例如下。

```
@POST
@Path("bio")
// 关注点 1: 资源方法入参
public String postStream(final InputStream is) throws FileNotFoundException,
IOException {
// 关注点 2: try-with-resources
    try (BufferedReader br = new BufferedReader(new InputStreamReader(is))) {
        StringBuilder result = new StringBuilder();
        String s = br.readLine();
        while (s != null) {
            result.append(s).append("\n");
            LOGGER.debug(s);
            s = br.readLine();
        }
        return result.toString(); // 关注点 3: 资源方法返回值
    }
}

@Test
public void testStream() {
// 关注点 4: 获取文件全路径
    final InputStream resource = getClass().getClassLoader().
getResourceAsStream("gua.txt");
    final Builder request = target(path).path("bio").request();
    Entity<InputStream> e = Entity.entity(resource, MediaType.TEXT_PLAIN_TYPE);
    final Response response = request.post(e, Response.class);
    result = response.readEntity(String.class);
// 关注点 5: 输出返回值内容
    LOGGER.debug(result);
}
```

在这段代码中，资源方法 `postStream()` 的输入参数类型是 `InputStream`，见关注点1；服务器端从中读取字节流，并最终释放该资源，见关注点2；返回值是 `String` 类型，内容是字节流信息，见关注点3；测试方法 `testStream()` 构建了 "gua.txt" 文件内容的字节流，作为请求实体提交，见关注点4；响应实体预期为 `String` 类型的 "gua.txt" 文件内容信息，见关注点5。

2.3.4 Reader 类型

本示例展示另一种 Java 读写模式，以字符流作为 REST 方法参数，示例如下。

```
@POST
@Path("cio")
// 关注点 1: 资源方法入参
public String postChars(final Reader r) throws FileNotFoundException, IOException {
    // 关注点 2: try-with-resources
    try (BufferedReader br = new BufferedReader(r)) {
        String s = br.readLine();
        if (s == null) {
            throw new Jaxrs2GuideNotFoundException("NOT FOUND FROM READER");
        }
        while (s != null) {
            LOGGER.debug(s);
            s = br.readLine();
        }
        return "reader";
    }
}

@Test
public void testReader() {
    // 关注点 3: 构建并提交 Reader 实例
    ClassLoader classLoader = getClass().getClassLoader();
    final Reader resource =
        new InputStreamReader(classLoader.getResourceAsStream("gua.txt"));
    final Builder request = target(path).path("cio").request();
    Entity<Reader> e = Entity.entity(resource, MediaType.TEXT_PLAIN_TYPE);
    final Response response = request.post(e, Response.class);
    result = response.readEntity(String.class);
    // 关注点 4: 输出返回值内容
    LOGGER.debug(result);
}
```

在这段代码中，资源方法 `postChars()` 的输入参数类型是 `Reader`，见关注点 1；服务器端从中读取字符流，并最终释放该资源，返回值是 `String` 类型，见关注点 2；测试方法 `testReader()` 构建了 "gua.txt" 文件内容的 `Reader` 实例，将字符流作为请求实体提交，见关注点 3；响应实体预期为 `String` 类型的 "gua.txt" 文件内容信息，见关注点 4。

2.3.5 XML 类型

XML 类型是使用最广泛的数据类型。Jersey 对 XML 类型的数据处理，支持 Java 领域的两大标准，即 JAXP (Java API for XML Processing, JSR-206) 和 JAXB (Java Architecture for XML Binding, JSR-222)。

阅读指南

本节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：`com.example.media.xml`。

1. JAXP 标准

JAXP 包含了 DOM、SAX 和 StAX 3 种解析 XML 的技术标准。

□ DOM 是面向文档解析的技术，要求将 XML 数据全部加载到内存，映射为树和结点模型以实现解析。

□ SAX 是事件驱动的流解析技术，通过监听注册事件，触发回调方法以实现解析。

□ StAX 是拉式流解析技术，相对于 SAX 的事件驱动推送技术，拉式解析使得读取过程可以主动推进当前 XML 位置的指针而不是被动获得解析中的 XML 数据。

对应的，JAXP 定义了 3 种标准类型的输入接口 Source (DOMSource, SAXSource, StreamSource) 和输出接口 Result (DOMResult, SAXResult, StreamResult)。Jersey 可以使用 JAXP 的输入类型作为 REST 方法的参数，示例代码如下。

```
@POST
@Path("stream")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public StreamSource getStreamSource(
    javax.xml.transform.stream.StreamSource streamSource) {
    // 关注点 1: 资源方法入参
    return streamSource;
}

@POST
@Path("sax")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
// 关注点 2: 支持 SAX 技术
public SAXSource getSAXSource(javax.xml.transform.sax.SAXSource saxSource) {
    return saxSource;
}

@POST
@Path("dom")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
// 关注点 3: 支持 DOM 技术
public DOMSource getDOMSource(javax.xml.transform.dom.DOMSource domSource) {
    return domSource;
}

@POST
@Path("doc")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
// 关注点 4: 支持 DOM 技术
public Document getDocument(org.w3c.dom.Document document) {
    return document;
}
```

在这段代码中，资源方法 `getStreamSource()` 使用 StAX 拉式流解析技术支持输入输出类型为 `StreamSource` 的请求，见关注点 1；`getSAXSource()` 方法使用 SAX 是事件驱动的

流解析技术支持输入输出类型为 SAXSource 的请求，见关注点 2；getDOMSource() 方法和 getDocument() 方法使用 DOM 面向文档解析的技术，支持输入输出类型分别为 DOMSource 和 Document 的请求，见关注点 3 和关注点 4。

2. JAXB 标准

JAXP 的缺点是需要编码解析 XML，这增加了开发成本，但对业务逻辑的实现并没有实质的贡献。JAXB 只需要在 POJO 中定义相关的注解（早期人们使用 XML 配置文件来做这件事），使其和 XML 的 schema 对应，无须对 XML 进行程序式解析，弥补了 JAXP 的这一缺点，因此本书推荐使用 JAXB 作为 XML 解析的技术。

JAXB 通过序列化和反序列化实现了 XML 数据和 POJO 对象的自动转换过程。在运行时，JAXB 通过编组（marshall）过程将 POJO 序列化成 XML 格式的数据，通过解编（unmarshall）过程将 XML 格式的数据反序列化为 Java 对象。JAXB 的注解位于 javax.xml.bind.annotation 包中，详情可以访问 JAXB 的参考实现网址是 <https://jaxb.java.net/tutorial>。

需要指出的是，从理论上讲，JAXB 解析 XML 的性能不如 JAXP，但使用 JAXB 的开发效率很高。笔者所在的开发团队使用 JAXB 解析 XML，从实践体会而言，笔者并不支持 JAXB 影响系统运行性能这样的观点。因为计算机执行的瓶颈在 IO，而无论使用哪种技术解析，XML 数据本身是一样的，区别仅在于解析手段。而 REST 风格以及敏捷思想的宗旨就是简单——开发过程简单化、执行逻辑简单化，因此如果连 XML 数据都趋于简单，JAXP 带来的性能优势就可以忽略不计了。综合考量，实现起来更简单的 JAXB 更适合做 REST 开发。

Jersey 支持使用 JAXBElement 作为 REST 方法参数的形式，也支持直接使用 POJO 作为 REST 方法参数的形式，后一种更为常用，示例代码如下。

```
@POST
@Path("/jaxb")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(JAXBElement<Book> bookElement) {
    Book book = bookElement.getValue();
    LOGGER.debug(book.getBookName());
    return book;
}

@POST
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(Book book) {
    LOGGER.debug(book.getBookName());
    return book;
}
```

以上 JAXP 和 JAXB 的测试如下所示，其传输内容是相同的，不同在于服务器端的

REST 方法定义的解析类型和返回值类型。

```
1 > Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="100"
bookName="TEST BOOK"/>
2 < Content-Length: 79
2 < Content-Type: text/html
<?xml version="1.0" encoding="UTF-8"?><book bookId="100" bookName="TEST
BOOK"/>
```

从测试结果可以看到，POJO 类的字段是作为 XML 的属性组织起来的，详见如下的图书实体类定义。

```
@XmlRootElement
public class Book implements Serializable {
// 关注点 1: JAXB 属性注解
    @XmlAttribute(name = "bookId")
    public Long getBookId() {
        return bookId;
    }
    @XmlAttribute(name = "bookName")
    public String getBookName() {
        return bookName;
    }
    @XmlAttribute(name = "publisher")
    public String getPublisher() {
        return publisher;
    }
}
```

(1) property 和 element

本例的 POJO 类 Book 的字段都定义为 XML 的属性 (property) 来组织，POJO 的字段也可以作为元素 (element) 组织，见关注点 1。如何定义通常取决于对接系统的设计。需要注意的是，如果 REST 请求的传输数据量很大，并且无须和外系统对接的场景，建议使用属性来组织 XML，这样可以极大地减少 XML 格式的数据包的大小。

(2) XML_SECURITY_DISABLE

Jersey 默认设置了 XMLConstants.FEATURE_SECURE_PROCESSING (<http://javax.xml.XMLConstants/feature/secure-processing>) 属性，当属性或者元素过多时，会报 “well-formedness error” 这样的警告信息。如果业务逻辑确实需要设计一个繁琐的 POJO，可以通过设置 MessageProperties.XML_SECURITY_DISABLE 参数值为 TRUE 来屏蔽。服务器端和客户端，示例代码如下。

```
@ApplicationPath("/")
public class XXXResourceConfig extends ResourceConfig {
    public XXXResourceConfig() {
        packages("xxx.yyy.zzz");
        property(MessageProperties.XML_SECURITY_DISABLE, Boolean.TRUE);
    }
}
```

```
    }  
}  
ClientConfig config = new ClientConfig();  
config.property(MessageProperties.XML_SECURITY_DISABLE, Boolean.TRUE);
```

2.3.6 JSON 类型

JSON 类型已经成为 Ajax 技术中数据传输的实际标准。Jersey 提供了 4 种处理 JSON 数据的媒体包。表 2-6 展示了 4 种技术对 3 种解析流派（基于 POJO 的 JSON 绑定、基于 JAXB 的 JSON 绑定以及低级的（逐字的）JSON 解析和处理）的支持情况。MOXy 和 Jackson 的处理方式相同，它们都不支持以 JSON 对象方式解析 JSON 数据，而是以绑定方式解析。Jettison 支持以 JSON 对象方式解析 JSON 数据，同时支持 JAXB 方式的绑定。JSON-P 就只支持 JSON 对象方式解析这种方式了。

表 2-6 Jersey 对 JSON 的处理方式列表

| 解析方式 \ JSON 支持包 | MOXy | JSON-P | Jackson | Jettison |
|-------------------------------------|------|--------|---------|----------|
| POJO-based JSON Binding | 是 | 否 | 是 | 否 |
| JAXB-based JSON Binding | 是 | 否 | 是 | 是 |
| Low-level JSON parsing & processing | 否 | 是 | 否 | 是 |

下面将介绍 MOXy、SON-P、Jackson 和 Jettison 这 4 种 Jersey 支持的 JSON 处理技术在 REST 式的 Web 服务开发中的使用。

1. 使用 MOXy 处理 JSON

MOXy 是 EclipseLink 项目的一个模块，其官方网站 <http://www.eclipse.org/eclipselink/moxy.php> 宣称 EclipseLink 的 MOXy 组件是使用 JAXB 和 SDO 作为 XML 绑定的技术基础。MOXy 实现了 JSR 222 标准（JAXB2.2）和 JSR 235 标准（SDO2.1.1），这使得使用 MOXy 的 Java 开发者能够高效地完成 Java 类和 XML 的绑定，所要花费的只是使用注解来定义它们之间的对应关系。同时，MOXy 实现了 JSR-353 标准（Java API for Processing JSON1.0），以 JAXB 为基础来实现对 JSR353 的支持。下面开始讲述使用 MOXy 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

2.3.6 节的 MOXy 示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.3.6-1.simple-service-moxy>。

(1) 定义依赖

MOXy 是 Jersey 默认的 JSON 解析方式，可以在项目中添加 MOXy 的依赖包来使用 MOXy。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-moxy</artifactId>
</dependency>
```

(2) 定义 Application

使用 Servlet3 可以不定义 web.xml 配置，否则请参考 1.6 节的讲述。

MOXy 的 Feature 接口实现类是 MoxyJsonFeature，默认情况下，Jersey 对其自动探测，无须在 Application 类或其子类显式注册该类。如果不希望 Jersey 这种默认行为，可以通过设置如下属性来禁用自动探测：CommonProperties.MOXY_JSON_FEATURE_DISABLE 两端禁用，ServerProperties.MOXY_JSON_FEATURE_DISABLE 服务器端禁用，ClientProperties.MOXY_JSON_FEATURE_DISABLE 客户端禁用。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        //property(org.glassfish.jersey.CommonProperties.MOXY_JSON_FEATURE_
DISABLE, true);
    }
}
```

(3) 定义资源类

接下来，我们定义一个图书资源类 BookResource，并在其中实现表述媒体类型为 JSON 的资源方法 getBooks()。支持 JSON 格式的表述的资源类定义如下。

```
@Path("books")
// 关注点 1: @Produces 注解和 @Consumes 注解上移到接口
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
    private static final HashMap<Long, Book> memoryBase;
    ...

    @GET
    // 关注点 2: 实现类方法无需再定义 @Produces 注解和 @Consumes 注解
    public Books getBooks() {
        final List<Book> bookList = new ArrayList<>();
        final Set<Map.Entry<Long, Book>> entries = BookResource.memoryBase.
entrySet();
        final Iterator<Entry<Long, Book>> iterator = entries.iterator();
        while (iterator.hasNext()) {
            final Entry<Long, Book> cursor = iterator.next();
            BookResource.LOGGER.debug(cursor.getKey());
            bookList.add(cursor.getValue());
        }
    }
}
```

```

    }
    final Books books = new Books(bookList);
    BookResource.LOGGER.debug(books);
    return books;
}
}

```

在这段代码中，资源类 `BookResource` 定义了 `@Consumes(MediaType.APPLICATION_JSON)` 和 `@Produces(MediaType.APPLICATION_JSON)`，表示该类的所有资源方法都使用 `MediaType.APPLICATION_JSON` 类型作为请求和响应的数据类型，见关注点 1；因此，`getBooks()` 方法上无须再定义 `@Consumes` 和 `@Produces`，见关注点 2。

如果 REST 应用处于多语言环境中，不要忘记统一开放接口的字符编码；如果统一开放接口同时供前端 jsonp 使用，不要忘记添加相关媒体类型，示例如下。

```
@Produces({"application/x-javascript;charset=UTF-8", "application/json;charset=UTF-8"})
```

在这段代码中，REST API 将支持 jsonp、json，并且统一字符编码为 UTF-8。

(4) 单元测试

JSON 处理的单元测试主要关注请求的响应中 JSON 数据的可用性、完整性和一致性。在本章使用的单元测试中，验证 JSON 处理无误的标准是测试的返回值是一个 Java 类型的实体类实例，整个请求处理过程中没有异常发生，测试代码如下。

```

public class JsonTest extends JerseyTest {
    private final static Logger LOGGER = Logger.getLogger(JsonTest.class);
    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testGettingBooks() {
        // 关注点 1: 在请求中定义媒体类型为 JSON
        Books books = target("books").request(MediaType.APPLICATION_JSON_TYPE).
            get(Books.class);
        for (Book book : books.getBookList()) {
            LOGGER.debug(book.getBookName());
        }
    }
}

```

在这段代码中，测试方法 `testGettingBooks()` 定义了请求资源的数据类型为 `MediaType.APPLICATION_JSON_TYPE` 来匹配服务器端提供的 REST API，其作用是定义请求的媒体类型为 JSON 格式的，见关注点 1。

(5) 集成测试

除了单元测试，我们使用 cURL 来做集成测试。首先启动本示例，然后输入如下所示

的命令。

```
curl http://localhost:8080/simple-service-moxy/api/books
curl -H "Content-Type: application/json" http://localhost:8080/simple-service-moxy/api/books
```

返回 JSON 格式的数据如下。

```
{ "bookList": { "book": [ { "bookId": 1, "bookName": "JSF2 和 RichFaces4 使用指南", "publisher": "电子工业出版社", "isbn": "9787121177378", "publishTime": "2012-09-01" }, { "bookId": 2, "bookName": "Java Restful Web Services 实战", "publisher": "机械工业出版社", "isbn": "9787111478881", "publishTime": "2014-09-01" }, { "bookId": 3, "bookName": "Java EE 7 精髓", "publisher": "人民邮电出版社", "isbn": "9787115375483", "publishTime": "2015-02-01" }, { "bookId": 4, "bookName": "Java Restful Web Services 实战 II", "publisher": "机械工业出版社" } ] } }
```

2. 使用 JSON-P 处理 JSON

JSON-P 的全称是 Java API for JSON Processing (Java 的 JSON 处理 API)，而不是 JSON with padding (JSONP)，两者只是名称相仿，用途大相径庭。JSON-P 是 JSR 353 标准规范，用于统一 Java 处理 JSON 格式数据的 API，其生产和消费的 JSON 数据以流的形式，类似 StAX 处理 XML，并为 JSON 数据建立 Java 对象模型，类似 DOM。而 JSONP 是用于异步请求中传递脚本的回调函数来解决跨域问题。下面开始讲述使用 JSON-P 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

2.3.6 节的 JSON-P 示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.3.6-2.simple-service-jsonp>。

(1) 定义依赖

使用 JSON-P 方式处理 JSON 类型的数据，需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-processing</artifactId>
</dependency>
```

(2) 定义 Application

使用 JSON-P 的应用，默认不需要在其 Application 中注册 JsonProcessingFeature，除非使用了如下设置。依次用于在服务器和客户端两侧去活 JSON-P 功能、在服务器端去活 JSON-P 功能、在客户端去活 JSON-P 功能。

☐ CommonProperties.JSON_PROCESSING_FEATURE_DISABLE

☐ ServerProperties.JSON_PROCESSING_FEATURE_DISABLE

❑ ClientProperties.JSON_PROCESSING_FEATURE_DISABLE

JsonGenerator.PRETTY_PRINTING 属性用于格式化 JSON 数据的输出, 当属性值为 TRUE 时, MesageBodyReader 和 MessageBodyWriter 实例会对 JSON 数据进行额外处理, 使得 JSON 数据可以格式化打印。该属性的设置在 Application 中, 见关注点 1, 示例代码如下。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        // 关注点 1: 配置 JSON 格式化输出
        property(JsonGenerator.PRETTY_PRINTING, true);
    }
}
```

(3) 定义资源类

资源类 BookResource 同上例一样定义了类级别的 @Consumes 和 @Produces, 媒体格式为 MediaType.APPLICATION_JSON, 资源类 BookResource 的示例代码如下。

```
@Path("books")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
    ...
    static {
        memoryBase = com.google.common.collect.Maps.newHashMap();
        // 关注点 1: 构建 JsonObjectBuilder 实例
        JsonObjectBuilder jsonObjectBuilder = Json.createObjectBuilder();
        // 关注点 2: 构建 JsonObject 实例
        JsonObject newBook1 = jsonObjectBuilder.add("bookId", 1)
            .add("bookName", "Java Restful Web Services 实战")
            .add("publisher", "机械工业出版社")
            .add("isbn", "9787111478881")
            .add("publishTime", "2014-09-01")
            .build();
        ...
    }
    @GET
    public JsonArray getBooks() {
        // 关注点 3: 构建 JsonArrayBuilder 实例
        final JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
        final Set<Map.Entry<Long, JsonObject>> entries =
            BookResource.memoryBase.entrySet();
        final Iterator<Entry<Long, JsonObject>> iterator = entries.iterator();
        while (iterator.hasNext()) {
            ...
        }
        // 关注点 4: 构建 JsonArray 实例
        JsonArray result = arrayBuilder.build();
        return result;
    }
}
```

在这段代码中, `JsonObjectBuilder` 用于构造 JSON 对象, 见关注点 1; `JsonArrayBuilder` 用于构造 JSON 数组对象, 见关注点 2; `JsonObject` 是 JSON-P 定义的 JSON 对象类, 见关注点 3; `JsonArray` 是 JSON 数组类, 见关注点 4。

(4) 单元测试

JSON-P 示例的单元测试需要关注 JSON-P 定义的 JSON 类型, 测试验收标准在前一小节 MOXY 的单元测试中已经讲述, 示例代码如下。

```
public class JsonTest extends JerseyTest {
    private final static Logger LOGGER = Logger.getLogger(JsonTest.class);
    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testGettingBooks() {
        // 关注点 1: 请求的响应类型为 JsonArray
        JsonArray books = target("books").request(MediaType.APPLICATION_JSON_TYPE).
            get(JsonArray.class);
        for (JsonValue jsonValue : books) {
            // 关注点 2: 强转 JsonValue 为 JsonObject
            JsonObject book = (JsonObject) jsonValue;
            LOGGER.debug(book.getString("bookName")); // 关注点 3: 打印输出测试结果
        }
    }
}
```

在这段代码片段中, `JsonArray` 是 `getBooks()` 方法的返回类型, `get()` 请求发出后, 服务器端对应的方法是 `getBooks()` 方法, 见关注点 1; `JsonValue` 类型是一种抽象化的 JSON 数据类型, 此处类型强制转化为 `JsonObject`, 见关注点 2; `getString()` 方法是将 `JsonObject` 对象的某个字段以字符串类型返回, 见关注点 3。

(5) 集成测试

使用 `cURL` 对本示例进行集成测试的结果如下所示, JSON 数据结果可以格式化打印输出。

```
curl http://localhost:8080/simple-service-jsonp/api/books
```

```
[
  {
    "bookId":1,
    "bookName":"Java Restful Web Services 实战",
    "publisher":"机械工业出版社",
    "isbn":"9787111478881",
    "publishTime":"2014-09-01"
  },
  {
```

```

        "bookId":2,
        "bookName":"JSF2 和 RichFaces4 使用指南 ",
        "publisher":" 电子工业出版社 ",
        "isbn":"9787121177378",
        "publishTime":"2012-09-01"
    },
    {
        "bookId":3,
        "bookName":"Java EE 7 精髓 ",
        "publisher":" 人民邮电出版社 ",
        "isbn":"9787115375483",
        "publishTime":"2015-02-01"
    },
    {
        "bookId":4,
        "bookName":"Java Restful Web Services 实战 II",
        "publisher":" 机械工业出版社 "
    }
]
curl http://localhost:8080/simple-service-jsonp/api/books/book?id=1
{
    "bookId":1,
    "bookName":"Java Restful Web Services 实战 ",
    "publisher":" 机械工业出版社 ",
    "isbn":"9787111478881",
    "publishTime":"2014-09-01"
}
curl -H "Content-Type: application/json" -X POST \
-d '{"bookName":"abc","publisher":"me"}' \
http://localhost:8080/simple-service-jsonp/api/books
{
    "bookId":23670621181527,
    "bookName":"abc",
    "publisher":"me"
}

```

3. 使用 Jackson 处理 JSON

Jackson 是一种流行的 JSON 支持技术，其源代码托管于 Github，地址是：<https://github.com/FasterXML/jackson>。Jackson 提供了 3 种 JSON 解析方式。

- ❑ 第一种是基于流式 API 的增量式解析 / 生成 JSON 的方式，读写 JSON 内容的过程是通过离散事件触发的，其底层基于 StAX API 读取 JSON 使用 `org.codehaus.jackson.JsonParser`，写入 JSON 使用 `org.codehaus.jackson.JsonGenerator`。
- ❑ 第二种是基于树型结构的内存模型，提供一种不变式的 `JsonNode` 内存树模型，类似 DOM 树。
- ❑ 第三种是基于数据绑定的方式，`org.codehaus.jackson.map.ObjectMapper` 解析，使用 JAXB 的注解。

下面开始讲述使用 Jackson 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

2.3.6 节的 Jackson 示例源代码地址: <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.3.6-3.simple-service-jackson>。

(1) 定义依赖

使用 Jackson 方式处理 JSON 类型的数据, 需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId>
</dependency>
```

(2) 定义 Application

使用 Jackson 的应用, 需要在其 Application 中注册 JacksonFeature。同时, 如果有必要根据不同的实体类做详细的解析, 可以注册 ContextResolver 的实现类, 示例代码如下。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        register(JacksonFeature.class);
        // 关注点 1: 注册 ContextResolver 的实现类 JsonContextProvider
        register(JsonContextProvider.class);
    }
}
```

在这段代码中, 注册了 ContextResolver 的实现类 JsonContextProvider, 用于提供 JSON 数据的上下文, 见关注点 1。有关 ContextResolver 详细信息参考 3.2 节。

(3) 定义 POJO

本例定义了 3 种不同方式的 POJO, 以演示 Jackson 处理 JSON 的多种方式。分别是 JsonBook、JsonHybridBook 和 JsonNoJaxbBook。第一种方式是仅用 JAXB 注解的普通的 POJO, 示例类 JsonBook 如下。

```
@XmlRootElement
@XmlType(propOrder = {"bookId", "bookName", "chapters"})
public class JsonBook {
    private String[] chapters;
    private String bookId;
    private String bookName;

    public JsonBook() {
        bookId = "1";
        bookName = "Java Restful Web Services 实战";
        chapters = new String[0];
    }
    ...
}
```

第二种方式是将 JAXB 的注解和 Jackson 提供的注解混合使用的 POJO，示例类 `JsonHybridBook` 如下。

```
// 关注点 1: 使用 JAXB 注解
@XmlRootElement
public class JsonHybridBook {
    // 关注点 2: 使用 Jackson 注解
    @JsonProperty("bookId")
    private String bookId;

    @JsonProperty("bookName")
    private String bookName;

    public JsonHybridBook() {
        bookId = "2";
        bookName = "Java Restful Web Services 实战";
    }
}
```

在这段代码中，分别使用了 JAXB 的注解 `javax.xml.bind.annotation.XmlRootElement`，见关注点 1，和 Jackson 的注解 `org.codehaus.jackson.annotate.JsonProperty`，见关注点 2，定义 XML 根元素和 XML 属性。

第三种方式是不使用任何注解的 POJO，示例类 `JsonNoJaxbBook` 如下。

```
public class JsonNoJaxbBook {
    private String[] chapters;
    private String bookId;
    private String bookName;
    public JsonNoJaxbBook() {
        bookId = "3";
        bookName = "Java Restful Web Services 使用指南";
        chapters = new String[0];
    }
    ...
}
```

这样的 3 种 POJO 如何使用 Jackson 处理来处理呢？我们继续往下看。

(4) 定义资源类

资源类 `BookResource` 用于演示 Jackson 对上述 3 种不同 POJO 的支持，示例代码如下。

```
@Path("books")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
    @Path("/emptybook")
    @GET
    // 关注点 1: 支持第一种方式的 POJO 类型
    public JsonBook getEmptyArrayBook() {
        return new JsonBook();
    }
    @Path("/hybirdbook")
```



```

    @GET
    // 关注点 2: 支持第二种方式的 POJO 类型
    public JsonHybridBook getHybirdBook() {
        return new JsonHybridBook();
    }
    @Path("/nojaxbbook")
    @GET
    // 关注点 3: 支持第三种方式的 POJO 类型
    public JsonNoJaxbBook getNoJaxbBook() {
        return new JsonNoJaxbBook();
    }
    .....

```

在这段代码中，资源类 `BookResource` 定义了路径不同的 3 个 GET 方法，返回类型分别对应上述的 3 种 POJO，见关注点 1 到 3。有了这样的资源类，就可以向其发送 GET 请求，并获取不同类型的 JSON 数据，以研究 Jackson 是如何支持这 3 种 POJO 的 JSON 转换。

(5) 上下文解析实现类

`JsonContextProvider` 是 `ContextResolver`（上下文解析器）的实现类，其作用是根据上下文提供的 POJO 类型，分别提供两种解析方式。第一种是默认的方式，第二种是混合使用 Jackson 和 Jaxb。两种解析方式的示例代码如下。

```

@Provider
public class JsonContextProvider implements ContextResolver<ObjectMapper> {
    final ObjectMapper d;
    final ObjectMapper c;
    public JsonContextProvider() {
        // 关注点 1: 实例化 ObjectMapper
        d = createDefaultMapper();
        c = createCombinedMapper();
    }
    private static ObjectMapper createCombinedMapper() {
        Pair ps = createIntrospector();
        ObjectMapper result = new ObjectMapper();
        result.setDeserializationConfig(ps);
        result.getDeserializationConfig().withAnnotationIntrospector(ps);
        result.setSerializationConfig(ps);
        result.getSerializationConfig().withAnnotationIntrospector(ps);
        return result;
    }
    private static ObjectMapper createDefaultMapper() {
        ObjectMapper result = new ObjectMapper();
        result.configure(Feature.INDENT_OUTPUT, true);
        return result;
    }
    private static Pair createIntrospector() {
        AnnotationIntrospector p = new JacksonAnnotationIntrospector();
        AnnotationIntrospector s = new JaxbAnnotationIntrospector();
        return new Pair(p, s);
    }
    @Override public ObjectMapper getContext(Class<?> type) {
        // 关注点 2: 判断 POJO 类型返回相应的 ObjectMapper 实例

```

```

    if (type == JsonHybridBook.class) {
        return c;
    } else {
        return d;
    }
}
}

```

在这段代码中，`JsonContextProvider` 定义并实例化了两种类型 `ObjectMapper`，见关注点 1；在实现接口方法 `getContext()` 中，通过判断当前 POJO 的类型，返回两种 `ObjectMapper` 实例之一，见关注点 2。通过这样的实现，当流程获取 JSON 上下文时，既可使用 Jackson 依赖包完成对相关 POJO 的处理。

(6) 单元测试

单元测试类 `BookResourceTest` 的目的是对支持上述 3 种 POJO 的资源地址发起请求并测试结果，示例如下。

```

public class BookResourceTest extends JerseyTest {
    private static final Logger LOGGER = Logger.getLogger(BookResourceTest.class);
    WebTarget booksTarget = target("books");
    @Override
    protected ResourceConfig configure() {
        // 关注点 1: 服务器端配置
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        ResourceConfig resourceConfig = new ResourceConfig(BookResource.class);
        // 关注点 2: 注册 JacksonFeature
        resourceConfig.register(JacksonFeature.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig config) {
        // 关注点 3: 注册 JacksonFeature
        config.register(new JacksonFeature());
        config.register(JsonContextProvider.class);
    }
    @Test
    // 关注点 4: 测试出参为 JsonBook 类型的资源方法
    public void testEmptyArray() {
        JsonBook book = booksTarget.path("emptybook").request(MediaType.APPLICATION_JSON).get(JsonBook.class);
        LOGGER.debug(book);
    }
    @Test
    // 关注点 5: 测试出参为 JsonHybridBook 类型的资源方法
    public void testHybrid() {
        JsonHybridBook book = booksTarget.path("hybirdbook").request(MediaType.APPLICATION_JSON).get(JsonHybridBook.class);
        LOGGER.debug(book);
    }
    @Test
    // 关注点 6: 测试出参为 JsonNoJaxbBook 类型的资源方法

```

```

public void testNoJaxb() {
    JsonNoJaxbBook book = booksTarget.path("nojaxbbook").request(MediaType.
APPLICATION_JSON).get(JsonNoJaxbBook.class);
    LOGGER.debug(book);
}
.....

```

在这段代码中，首先要在服务器端注册支持 Jackson 功能，见关注点 2；同时在客户端也要注册支持 Jackson 功能并注册 JsonContextProvider，见关注点 3；该测试类包含了用于测试 3 种类型 POJO 的测试用例，见关注点 4 到 6；注意，configure() 方法是覆盖测试服务实例行为，configureClient() 方法是覆盖测试客户端实例行为，见关注点 1。

(7) 集成测试

使用 cURL 对本例进行集成测试，结果如下所示。

```
curl http://localhost:8080/simple-service-jackson/api/books
```

```

{
  "bookList" : [ {
    "bookId" : 1,
    "bookName" : "JSF2 和 RichFaces4 使用指南 ",
    "isbn" : "9787121177378",
    "publisher" : " 电子工业出版社 ",
    "publishTime" : "2012-09-01"
  }, {
    "bookId" : 2,
    "bookName" : "Java Restful Web Services 实战 ",
    "isbn" : "9787111478881",
    "publisher" : " 机械工业出版社 ",
    "publishTime" : "2014-09-01"
  }, {
    "bookId" : 3,
    "bookName" : "Java EE 7 精髓 ",
    "isbn" : "9787115375483",
    "publisher" : " 人民邮电出版社 ",
    "publishTime" : "2015-02-01"
  }, {
    "bookId" : 4,
    "bookName" : "Java Restful Web Services 实战 II",
    "isbn" : null,
    "publisher" : " 机械工业出版社 ",
    "publishTime" : null
  } ]
}

```

```
curl http://localhost:8080/simple-service-jackson/api/books/emptybook
```

```

{
  "chapters" : [ ],
  "bookId" : "1",
  "bookName" : "Java Restful Web Services 实战 "
}

```

```
curl http://localhost:8080/simple-service-jackson/api/books/hybridbook
```

```
{ "JsonHybridBook": { "bookId": "2", "bookName": "Java Restful Web Services 实战" } }
curl http://localhost:8080/simple-service-jackson/api/books/nojaxbbook

{
  "chapters" : [ ],
  "bookId" : "3",
  "bookName" : "Java Restful Web Services 实战 "
}
```

4. 使用 Jettison 处理 JSON

Jettison 是一种使用 StAX 来解析 JSON 的实现。项目地址是：<http://jettison.codehaus.org>。Jettison 项目起初用于为 CXF 提供基于 JSON 的 Web 服务，在 XStream 的 Java 对象的序列化中也使用了 Jettison。Jettison 支持两种 JSON 映射到 XML 的方式。Jersey 默认使用 MAPPED 方式，另一种叫做 BadgerFish 方式。

下面开始讲述使用 Jettison 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

2.3.6 节的 Jettison 示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.3.6-4.simple-service-jettison>。

(1) 定义依赖

使用 Jettison 方式处理 JSON 类型的数据，需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jettison</artifactId>
</dependency>
```

(2) 定义 Application

使用 Jettison 的应用，需要在其 Application 中注册 JettisonFeature。同时，如果有必要根据不同的实体类做详细的解析，可以注册 ContextResolver 的实现类，示例代码如下。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        // 关注点 1: 注册 JettisonFeature 和 ContextResolver 的实现类 JsonContextResolver
        register(JettisonFeature.class);
        register(JsonContextResolver.class);
    }
}
```

在这段代码中，注册了 Jettison 功能 JettisonFeature 和 ContextResolver 的实现类 JsonContextResolver，以便使用 Jettison 处理 JSON，见关注点 1。

(3) 定义 POJO

本例定义了两个类名不同、内容相同的 POJO (JsonBook 和 JsonBook2), 用以演示 Jettison 对 JSON 数据以 JETTISON_MAPPED (default notation) 和 BADGERFISH 两种不同方式的处理情况。

```
@XmlElement
public class JsonBook {
    private String bookId;
    private String bookName;
    public JsonBook() {
        bookId = "1";
        bookName = "Java Restful Web Services 实战 ";
    }
    ...
}
```

(4) 定义资源类

资源类 BookResource 为两种 JSON 方式提供了资源地址, 示例如下。

```
@Path("books")
public class BookResource {
    ...
    @Path("/jsonbook")
    @GET
    // 关注点 1: 返回类型为 JsonBook 的 GET 方法
    public JsonBook getBook() {
        final JsonBook book = new JsonBook();
        BookResource.LOGGER.debug(book);
        return book;
    }
    @Path("/jsonbook2")
    @GET
    // 关注点 2: 返回类型为 JsonBook2 的 GET 方法
    public JsonBook2 getBook2() {
        final JsonBook2 book = new JsonBook2();
        BookResource.LOGGER.debug(book);
        return book;
    }
}
```

在这段代码中, 资源类 BookResource 定义了路径不同的两个 GET 方法, 返回类型分别是 JsonBook 和 JsonBook2, 见关注点 1 和 2。有了这样的资源类, 就可以向其发送 GET 请求, 并获取不同类型的 JSON 数据, 以研究 Jettison 是如何处理 JETTISON_MAPPED 和 BADGERFISH 两种不同格式的 JSON 数据的。

(5) 上下文解析实现类

JsonContextResolver 实现了 ContextResolver 接口, 示例如下。

```
@Provider
public class JsonContextResolver implements ContextResolver<JAXBContext> {
```



```

private final JAXBContext context1;
private final JAXBContext context2;
@SuppressWarnings("rawtypes")
public JsonContextResolver() throws Exception {
    Class[] clz = new Class[] { JsonBook.class, JsonBook2.class, Books.class, Book.class };
    // 关注点 1: 实例化 JettisonJaxbContext
    this.context1 = new JettisonJaxbContext(JettisonConfig.DEFAULT, clz);
    this.context2 = new JettisonJaxbContext(JettisonConfig.badgerFish().build(), clz);
}
@Override
public JAXBContext getContext(Class<?> objectType) {
    // 关注点 2: 判断 POJO 类型返回相应的 JAXBContext 实例
    if (objectType == JsonBook2.class) {
        return context2;
    } else {
        return context1;
    }
}
}

```

在这段代码中，JsonContextResolver 定义了两种 JAXBContext 分别使用 MAPPED 方式或者 BadgerFish 方式，见关注点 1。这两种方式的参数信息来自 Jettison 依赖包的 JettisonConfig 类。在实现接口方法 getContext() 中，根据不同的 POJO 类型，返回两种 JAXBContext 实例之一，见关注点 2。通过这样的实现，当流程获取 JSON 上下文时，既可使用 Jettison 依赖包完成对相关 POJO 的处理。

(6) 单元测试

单元测试类 BookResourceTest 的目的是对支持上述两种 JSON 方式的资源地址发起请求并测试结果，示例如下。

```

public class BookResourceTest extends JerseyTest {
    private static final Logger LOGGER = Logger.getLogger(BookResourceTest.class);
    @Override
    protected ResourceConfig configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        ResourceConfig resourceConfig = new ResourceConfig(BookResource.class);
        // 关注点 1: 注册 JettisonFeature 和 JsonContextResolver
        resourceConfig.register(JettisonFeature.class);
        resourceConfig.register(JsonContextResolver.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig config) {
        // 关注点 2: 注册 JettisonFeature 和 JsonContextResolver
        config.register(new JettisonFeature()).register(JsonContextResolver.class);
    }
    @Test
    public void testJsonBook() {

```

```

// 关注点 3: 测试返回类型为 JsonBook 的 GET 方法
JsonBook book = target("books").path("jsonbook")
    .request(MediaType.APPLICATION_JSON).get(JsonBook.class);
LOGGER.debug(book);
//{"jsonBook":{"bookId":1,"bookName":"abc"}}
}
@Test
public void testJsonBook2() {
    // 关注点 4: 测试返回类型为 JsonBook2 的 GET 方法
    JsonBook2 book = target("books").path("jsonbook2")
        .request(MediaType.APPLICATION_JSON).get(JsonBook2.class);
    LOGGER.debug(book);
    //{"jsonBook2":{"bookId":{"$":"1"},"bookName":{"$":"abc"}}}
}
...
}

```

在这段代码中, 首先要在服务器和客户端两侧注册 Jettison 功能和 JsonContextResolver, 见关注点 1 和 2。该测试类包含了用于测试两种格式 JSON 数据的测试用例, 见关注点 3 和 4。

(7) 集成测试

使用 cURL 对本例进行集成测试, 结果如下所示。可以看到 Mapped 和 Badgerfish 两种方式的 JSON 数据内容不同。

```
curl http://localhost:8080/simple-service-jettison/api/books
```

```

{"books":{"bookList":[{"@bookId":"1","@bookName":"JSF2 和 RichFaces4
使用指南","@publisher":"电子工业出版社","isbn":"9787121177378","publishTime":"
2012-09-01"}, {"@bookId":"2","@bookName":"Java Restful Web Services 实 战 ","@
publisher":"机械工业出版社","isbn":"9787111478881","publishTime":"2014-09-01"}, {
"@bookId":"3","@bookName":"Java EE 7 精髓","@publisher":"人民邮电出版社","isbn
":"9787115375483","publishTime":"2015-02-01"}, {"@bookId":"4","@bookName":"Java
Restful Web Services 实战 II","@publisher":"机械工业出版社"}]}}}

```

Jettison mapped notation

```
curl http://localhost:8080/simple-service-jettison/api/books/jsonbook
```

```
{"jsonBook":{"bookId":1,"bookName":"Java Restful Web Services 实战"}}
```

Badgerfish notation

```
curl http://localhost:8080/simple-service-jettison/api/books/jsonbook2
```

```

{"jsonBook2":{"bookId":{"$":"1"},"bookName":{"$":"Java Restful Web Services 实
战" }}}

```

最后简要介绍一下 Atom 类型。

Atom 是一种基于 XML 的文档格式, 该格式的标准定义在 IETF RFC 4287 (Atom Syndication Format, Atom 联合格式), 其推出的目的是用来替换 RSS。AtomPub 是基于 Atom 的发布协议, 定义在 IETF RFC 5023 (Atom Publishing Protocol)。

Jersey2 没有直接引入支持 Atom 格式的媒体包, 但 Jersey1.x 中包含 jersey-atom 包。

这说明 Jersey 的基本架构可以支持基于 XML 类型的数据，这种可插拔的媒体包支持对于 Jersey 本身更具灵活性，对使用 Jersey 的 REST 服务更具可扩展性。

2.4 连通性

REST 的一个重要的特性就是连通性。Web Link 和 HATEOAS 以不同方式实现了 REST 式服务的联通性。

❑ Web Link 定义在 IETF RFC 5988 (Web Linking)，是通过在 HTTP 头中定义链接信息，以描述当前页面与链接页面之间的关系。Web Link 是一种过渡型链接 (Transitional Links)。JAX-RS 2.0 引入了 `javax.ws.rs.core.Link` 类，用来处理 Web Link 的表述。

❑ HATEOAS (Hypermedia as the Engine of Application State，超媒体作为应用程序状态引擎)。HATEOAS 的形式是包含链接信息的超媒体文档，HATEOAS 的核心是“引擎”，该引擎的目的是通过请求的响应实体将超媒体信息返回给客户端，超媒体信息可以告诉用户，如果接下来选择去往某个链接 (或者链接列表中的某个链接)，应用的状态就会如超媒体描述的那样发生转变。HATEOAS 是一种结构型链接 (Structural Links)。Jersey2 中可以使用 XML 实现 HATEOAS 的结构要求。

下面讲述在 Jersey 中如何实现 Web Link 和 HATEOAS 这两种 REST 连通性实践方式。

阅读指南

本节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：`com.example.link`。

2.4.1 过渡型链接

Web Link 通过使用 HTTP 的头信息来传递操作链接，在 Jersey 中使用 `javax.ws.rs.core.Link` 类可以非常简洁地实现支持 Web Link 的资源类，示例代码如下。

```
@Path("weblink-resource")
public class WebLinkResource {
    @Context
    UriInfo uriInfo;
```

```

@POST
@Produces(MediaType.APPLICATION_XML)
@Consumes({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public Response saveBook(final Book book) {
    final long newId = System.nanoTime();
    book.setBookId(newId);
    LinkCache.map.put(newId, book);
    // 关注点 1: 通过 UriInfo 实例获取资源路径
    final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
    final URI location = ub.path("") + newId).build();
    // 关注点 2: 通过模板获取资源路径
    final String uriTemplate = "http://{host}:{port}/{path}/{param}";
    final URI location2 = UriBuilder.fromUri(uriTemplate)
        .resolveTemplate("host", "localhost").resolveTemplate("port", "9998")
        .resolveTemplate("path", "weblink-resource")
        .resolveTemplate("param", newId).build();
    // 关注点 3: 通过模板方法获取资源路径
    final UriBuilder ub3 = uriInfo.getAbsolutePathBuilder();
    final URI location3 = ub3.scheme("http").host("localhost").port(9998)
        .path("weblink-resource").path("") + newId).build();
    // 关注点 4: 为响应实例添加路径信息
    return Response.created(location).link(location2, "view1")
        .link(location3, "view2").entity(book).build();
}
}

```

在这段代码中，使用了 3 种方式构建 URI 实例。第一种方式是通过调用 UriInfo 实例的 getAbsolutePathBuilder() 方法可以获取当前请求的绝对路径，然后基于此路径添加资源 id 信息，见关注点 1；第二种方式是 UriBuilder 提供路径模板，然后链式调用 resolveTemplate() 方法传递并解析模板参数，最后通过 UriBuilder 的 build 方法生成 URI 实例，见关注点 2；第三种方式和第二种类似，不同的是模板信息被具体方法替代。最后，这 3 个与 Link 相关的 URI 实例由 Response 构建，作为返回值响应给客户端，见关注点 4。

2.4.2 结构型链接

HATEOAS 用以代替聚集数据并避免描述膨胀，通常使用 Atom 格式在实体字段中提供链接信息。本例使用 XML 格式来支持 HATEOAS，折中的设计是在 POJO 中额外定义一个链接字段。支持 HATEOAS 的资源类示例如下。

```

@Path("hateoas-resource")
public class HATEOASResource {
    @Context
    UriInfo uriInfo;

    @POST

```

```

@Produces({ MediaType.APPLICATION_XML })
@Consumes({ MediaType.APPLICATION_XML })
public BookWrapper saveBook(final Book book) {
    final long newId = System.nanoTime();
    book.setBookId(newId);
    LinkCache.map.put(newId, book);
    // 关注点 1: 通过 UriInfo 实例获取资源路径
    final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
    final URI uri = ub.path("/") + newId).build();
    BookWrapper b = new BookWrapper();
    b.setBook(book);
    // 关注点 2: 将资源路径赋值给资源实体
    b.setLink(uri.toString());
    return b;
}
}

```

在这段代码中，URI 实例由上下文 UriInfo 中获取的绝对路径和资源 ID 组成，见关注点 1；该链接信息被赋值到 POJO 实例的 link 属性中，以实现 HATEOAS，见关注点 2。

阅读指南

REST 连通性的实践手段非常多，推荐读者从成熟的产品中学习其设计。如果有可能，这里推荐 Jenkins 和 RallyDev 两个敏捷开发中常用的平台，它们提供了比较舒适的连通性设计。比如在 RallyDev 中，为一个测试用例结果添加测试用例属性（该属性是必填项），其内容并不是对应测试用例的实例，而是该测试用例的引用地址字符串。这样的设计不但减少了网络传输的负载，还方便在调试和维护时排错。

2.5 处理响应

REST 的响应处理结果应包括响应头中 HTTP 状态码，响应实体中媒体参数类型和返回值类型，以及异常情况处理。JAX-RS2 支持 4 种返回值类型的响应，分别是无返回值、返回 Response 类实例、返回 GenericEntity 类实例和返回自定义类实例。如下，逐一讲述这 4 种返回值类型。

阅读指南

本节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：com.example.response。

2.5.1 返回类型

1. void

在返回值类型是 `void` 的响应中，其响应实体为空，HTTP 状态码为 204。在前面的 DELETE 方法讲述中已经介绍过，再来看一下这种类型的资源方法。

```
@DELETE
@Path("/{s}")
// 关注点 1: 无返回值的 DELETE 方法
public void delete(@PathParam("s") final String s) {
    LOGGER.debug(s);
}
```

因为 `delete` 操作无须返回更多的关于资源表述的信息，因此该方法没有返回值，即返回值类型为 `void`，见关注点 1。

2. Response

在返回值类型为 `Response` 的响应中，响应实体为 `Response` 类的 `entity()` 方法定义的实体类实例。如果该内容为空，则 HTTP 状态码为 204，否则 HTTP 状态码为 200 OK，示例代码如下。

```
@POST
@Path("/c")
public Response get(final String s) {
    LOGGER.debug(s);
    //Response.noContent().build();
    // 关注点 1: 构建无返回值的响应实例
    return Response.ok().entity("char[]" + s).build();
}
```

在这段代码中，`Response` 首先定义了 HTTP 的状态码为 `ok`，然后填充实体信息，最后调用 `build()` 方法构建 `Response` 实例，见关注点 1。

3. GenericEntity

通用实体类型作为返回值的情况并不常用。其形式是构造一个统一的实体实例并将其返回，实体实例作为第一个参数、该实体类型作为第二个参数，示例代码如下。

```
@POST
@Path("/b")
public String get(final byte[] bs) {
    for (final byte b : bs) {
        LOGGER.debug(b);
    }
    return "byte[]" + new String(bs);
}
/*
public GenericEntity<String> get(final byte[] bs) {
```

```

        for (final byte b : bs) {
            LOGGER.debug(b);
        }
        // 关注点 1: 构建 GenericEntity 实例
        return new GenericEntity<String>("byte[]" + new String(bs), String.class);
    }
    */

```

在这段代码中, GenericEntity 的第一个是由 byte 数组实例作为参数构建的字符串实例, 第二个参数是字符串类, 见关注点 1。

4. 自定义类型

JDK 中的类 (比如 File、String 等) 都可以作为返回值类型, 更常用的是返回自定义的 POJO 类型, 前述多个例子就是这样做的, 再来看一个示例。

```

@POST
@Path("/f")
// 关注点 1: GET 方法的返回类型为 File
public File get(final File f) throws FileNotFoundException, IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(f))) {
        String s;
        do {
            s = br.readLine();
            LOGGER.debug(s);
        } while (s != null);
        return f;
    }
}
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces(MediaType.APPLICATION_XML)
// 关注点 2: POST 方法的返回值是自定义类 Book
public Book getEntity(Book book) {
    LOGGER.debug(book.getBookName());
    return book;
}
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces(MediaType.APPLICATION_XML)
// 关注点 3: POST 方法的返回值是自定义类 Book
public Book getEntity(JAXBElement<Book> bookElement) {
    Book book = bookElement.getValue();
    LOGGER.debug(book.getBookName());
    return book;
}

```

在这段代码中, 返回值类型有来自 JDK 的 File 类型, 见关注点 1, 也有自定义的 POJO 类型, 见关注点 2 和关注点 3。

2.5.2 处理异常

实现 REST 的资源方法时应使其具有良好的异常处理能力, 这包括异常的定义和错误

状态码的正确返回。

1. 处理状态码

首先通过表 2-7 了解下 REST 中常用的 HTTP 状态码，应当在处理异常的同时，为 REST 请求的客户端提供对应的错误码。

表 2-7 HTTP 常用状态码列表

| 状态码 | 含 义 |
|---------------------------|-------------------------------------|
| 200 OK | 服务器正常响应 |
| 201 Created | 创建新实体，响应头 Location 指定访问该实体的 URL |
| 202 Accepted | 服务器接受请求，处理尚未完成。可用于异步处理机制 |
| 204 No Content | 服务器正常响应，但响应实体为空 |
| 301 Moved Permanently | 请求资源的地址发生永久变动，响应头 Location 指定新的 URL |
| 302 Found | 请求资源的地址发生临时变动 |
| 304 Not Modified | 客户端缓存资源依然有效 |
| 400 Bad Request | 请求信息出现语法错误 |
| 401 Unauthorized | 请求资源无法授权给未验证用户 |
| 403 Forbidden | 请求资源未授权当前用户 |
| 404 Not Found | 请求资源不存在 |
| 405 Method Not Allowed | 请求方法不匹配 |
| 406 Not Acceptable | 请求资源的媒体类型不匹配 |
| 500 Internal Server Error | 服务器内部错误，意外终止响应 |
| 501 Not Implemented | 服务器不支持当前请求 |

JAX-RS2 规定的 REST 式的 Web 服务的基本异常类型为运行时异常 `WebApplicationException` 类。该类包含 3 个主要的子类分别对应如下内容。

□ HTTP 状态码为 3xx 的重定向类 `RedirectionException`;

□ HTTP 状态码为 4xx 的请求错误类 `ClientErrorException`;

□ HTTP 状态码为 5xx 的服务器错误类 `ServerErrorException`。

它们各自的子类对照 HTTP 状态码再细分，比如常见的 HTTP 状态码 404 错误，对应的错误类为 `NotFoundException`，如图 2-4 所示。

除了 Jersey 提供的标准异常类型，我们也可以根据业务需要自定义相关的业务异常类，示例如下。

```
// 关注点 1: 定义 WebApplicationException 接口实现类
public class Jaxrs2GuideNotFoundException extends WebApplicationException {
    public Jaxrs2GuideNotFoundException() {
// 关注点 2: 定义 HTTP 状态
        super(javax.ws.rs.core.Response.Status.NOT_FOUND);
    }
}
```

```

public Jaxrs2GuideNotFoundException(String message) {
    super(message);
}
}

```

在这段代码中，Jaxrs2GuideNotFoundException 类继承自 JAX-RS2 的 WebApplication-Exception 类，见关注点 1。其默认构造子提供了 HTTP 状态码，其值为 Response.Status.NOT_FOUND，见关注点 2。

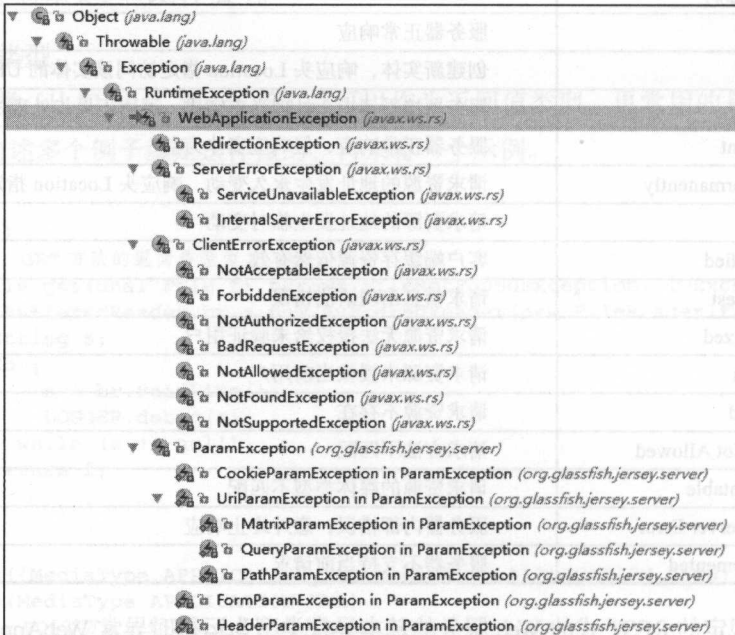


图 2-4 Jersey 定义的异常类型

2. ExceptionMapper

Jersey 框架为我们提供了更为通用的异常处理方式。通过实现 ExceptionMapper 接口并使用 @Provider 注解将其定义为一个 Provider，可以实现通用的异常的面向切面处理，而非针对某一个资源方法的异常处理，示例如下。

```

@Provider
public class EntityNotFoundException
implements ExceptionMapper<Jaxrs2GuideNotFoundException>{
    // 关注点 1: 定义 ExceptionMapper 接口实现类
    @Override
    public Response toResponse(final Jaxrs2GuideNotFoundException ex) {
        // 关注点 2: 拦截并返回新的响应实例
        return Response.status(404).entity(ex.getMessage()).type("text/plain").build();
    }
}

```

在这段代码中，EntityNotFoundMapper 实现了 ExceptionMapper 接口，并提供了泛型类型为前述刚定义的 Jaxrs2GuideNotFoundException 类，见关注点 1；当响应中发生了 Jaxrs2GuideNotFoundException 类型的异常，响应流程就会被拦截并补充 HTTP 状态码和异常消息，以文本作为媒体格式返回给客户端，见关注点 2。

2.6 内容协商

一个资源可以有不同格式的表述，表述（即响应实体）的内容是人类可识别的信息，服务器很难使用一种表述来适应所有用户。conneg（HTTP Content Negotiation，内容协商）是指在服务器提供的多种表述中，为特定的请求选择最好的一种表述的处理过程。那么什么是最好，又怎样做到最好呢？服务器和客户端/浏览器之间往复通信来协商用于交换数据的内容格式等信息，达成一致即为最好。内容协商定义在 RFC2616 的第 12 节（<http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html>）。

客户端/浏览器通过使用 HTTP Accept、Accept-Charset、Accept-Language 和 Accept-Encoding 头来定义接收头的信息，将其所期待的格式或 MIME 类型告知服务器，服务器根据协商算法，返回客户端/浏览器可接受的数据信息。内容协商不只是数据格式协商，还包括语言、编码、字符集等信息。Accept 用于数据类型协商；Accept-Language 用于语言协商；Accept-Charset 用于字符集协商；Accept-Encoding 用于压缩算法协商。

JAX-RS2 对内容协商的支持，是通过 @Produces 实现的，其他协商没有从架构上提供支持，可以通过编码从请求头中获取信息并处理。

阅读指南

本节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/2.simple-service-3>。

相关包：com.example.conneg。

2.6.1 @Produces 注解

注解 @Produces 用于定义方法的响应实体的数据类型，可以定义一个或多个，同时可以为每种类型定义质量因素（qualityfactor）。质量因素是取值范围从 0 到 1 的小数值。如果不定义质量因素，那么该类型的质量因素默认为 1。我们将结合示例深入了解 @Produces 注解对媒体类型的影响，示例代码如下。


```

@Path("conneg-resource")
public class ConnegResource {
    @GET
    @Path("{id}")
    // 关注点 1: 媒体类型为 XML
    @Produces(MediaType.APPLICATION_XML)
    public Book getJaxbBook(@PathParam("id") final Long bookId) {
        return new Book(bookId);
    }

    @GET
    @Path("{id}")
    // 关注点 2: 媒体类型为 JSON
    @Produces(MediaType.APPLICATION_JSON)
    public Book getJsonBook(@PathParam("id") final Long bookId) {
        return new Book(bookId);
    }
}

```

在这段代码中, `getJaxbBook()` 和 `getJsonBook()` 是同等质量因素、资源地址相同的两个 GET 方法, 一个定义响应实体格式为 XML, 一个定义响应实体格式为 JSON, 见关注点 1 和 2。那么对同一个资源的访问, JAX-RS2 该如何选择处理方法呢? 如果请求中明确定义可接受的数据类型为两者之一, 处理方法应该是定义相应数据类型的方法。如果两者都定义了, 处理方法应该是质量因素高的方法。如果两者都定义, 而且数据类型的质量因素是相等的或者没有定义 `Accept`, XML 的方法会被优先选择。

客户端明确表述格式为 XML, Jersey 通过内容协商, 会选择 `getJaxbBook()` 作为相应的资源方法来处理该请求。其测试代码如下所示。

```

WebTarget path = target("conneg-resource").path("123");
Builder request = path.request(MediaType.APPLICATION_XML_TYPE);
Book book = request.get(Book.class);

```

```

1 > GET http://localhost:9998/conneg-resource/123
1 > Accept: application/xml
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="123"/>

```

接下来, 测试一个稍微复杂的内容协商。客户端明确表述格式的质量因素 JSON 高于 XML, Jersey 会选择资源方法 `getJsonBook()` 来处理请求。示例代码如下所示。

```

WebTarget path = target("conneg-resource").path("123");
Builder request = path.request();
request.header("Accept", "application/xml;q=0.1,application/json;q=0.2");
Book book = request.get(Book.class);
...1 > GET http://localhost:9998/conneg-resource/123 1 > Accept: application/
xml;q=0.1,application/json;q=0.2 2 < Content-Type: application/json
{"bookId":123}

```

现在我们清楚了两个同等方法的场景, 再来看一个方法中多种数据类型的场景。示例

代码如下。

```
...java
@GET
@Produces({ "application/json; qs=.9", "application/xml; qs=.5" })
@Path("book/{id}")
public Book getBook(@PathParam("id") final Long bookId) {
    return new Book(bookId);
}
```

在这段代码中，getBook() 方法定义了 XML 和 JSON 两种表述数据类型，XML 的质量因素是 0.5（0 可以省略），JSON 的是 0.9。

因此，可以推断，如果客户端请求中，明确接收的数据类型是两者之一，响应实体使用指定类型。如果没有定义或者两者都定义且 JSON 的质量因素大于或者等于 XML，则返回 JSON 类型。还有一种用例是，两者都定义但 JSON 的质量因素小于 XML，该如何处理请求方法呢？答案是：内容协商的结果按照客户端的喜好选择响应实体的数据类型，即选择 XML 格式。

其测试代码如下所示，客户端明确表述格式 XML 优于 JSON，虽然服务器端定义的资源方法中 JSON 的质量因素高，但 Jersey 会根据客户端的喜好，选择了 XML 格式作为表述的格式返回。

```
WebTarget path = target("conneg-resource").path("book").path("123");
Builder request = path.request();
request.header("Accept", "application/xml;q=0.7,application/json;q=0.2");
Book book = request.get(Book.class);

1 > GET http://localhost:9998/conneg-resource/book/123
1 > Accept: application/xml;q=0.7,application/json;q=0.2
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="123"/>
```

2.6.2 @Consumes 注解

注解 @Consumes 用于定义方法的请求实体的数据类型，和 @Produces 不同的是，@Consumes 的数据类型的定义只用于 JAX-RS2 匹配请求处理的方法，不做内容协商使用。如果匹配不到，服务器会返回 HTTP 状态码 415 (Unsupported Media Type)，示例代码如下。

```
@POST
// 关注点 1: @Consumes 注解定义了 XML 和 JSON 两种格式
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(Book book) {
    LOGGER.debug(book.getBookName());
    return book;
}

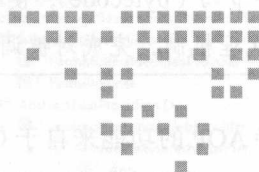
final Builder request = target(path).request();
```

```
// 关注点 2
final Book result = request.post(
    Entity.entity(book, MediaType.APPLICATION_XML), Book.class);
```

在这段代码中, `getEntity()` 方法定义了 `@Consumes` 媒体类型为 XML 格式和 JSON 格式, 见关注点 1; 那么, 在客户端请求中, 如果请求实体的数据类型定义是两者之一, 该方法会被选择为处理请求的方法, 否则查找是否有定义为相应数据类型的方法, 如果没有抛出 `javax.ws.rs.NotSupportedException` 异常, 则使用该方法处理请求, 见关注点 2。

2.7 本章小结

本章是 REST 理论和 Jersey 实践的核心章节, 详细讲述了 HTTP 方法与 REST API 的统一接口设计、URI 的 REST 风格设计, 并逐个讲述了 JAX-RS2 定义的注解如何支持资源定位, 还对 Jersey 对各种表述类型的支持和实现、Jersey 对 REST 连通性的两种实现、REST 资源方法对响应的处理以及 Jersey 对内容协商的支持和实现进行了讲述。



REST 请求处理

设计良好的 REST API 除了要符合关于统一接口和资源定位等要求，还要详细考虑通用的请求处理过程中每个步骤的特殊处理，并设计出符合业务规范的处理流程。本章将深入 REST 请求处理过程中的扩展点，并讲述如何对其实现。

阅读指南

本章节示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/3.jaxrs2-handle>。

3.1 Jersey 的 AOP 机制

AOP 对增强 REST 服务的功能性、安全性和可扩展性等方面都具有深远意义，因此，完整的 REST 风格的框架都从容器级别支持 AOP 功能。Jersey 自身支持 AOP，可以不依赖于 Spring 等支持 AOP 的框架。

阅读指南

AOP (Aspect Oriented Programming, 面向切面编程) 的典型应用场景有权限管理、日志记录、统计记录、事务以及异常处理等。其实现原理是代理被调用的方法，在其被执行的方法前后，增加额外业务功能。AOP 的实现机制是通过注解或者 XML 配置，依据这些配

置，动态生成字节码 (bytecode)，使被调用代码对应的字节码被环绕注入新的功能；或者使用 Java 的动态代理机制，完成对被调用方法的增强。

Jersey 支持 AOP 的功能来自于 GlashFish 项目集的 HK2 项目 (参见 1.4 节)。Jersey 通用包 jersey-common 依赖 HK2 (轻量级 DI 架构)，包括 hk2-api 和 hk2-locator。

其中 hk2-locator 依赖于 javax.inject 包、asm-all-repackaged 包和 cglib 包。从这些包名不难看出，hk2-locator 包是个纯粹玩 AOP 的。javax.inject 包出自 Java 依赖注入规范 (JSR-330)，cglib 包是 Spring 用户熟识的动态代码生成工具，是 Spring 两种 AOP 实现方式的一种，其底层依赖于 ASM (<http://asm.ow2.org>)。ASM 来自于开源软件国际联盟 OW2 (<http://www.ow2.org>)，OW2 的前身是来自法国的 ObjectWeb 和来自中国的 OrientWare 两个中间件开源组织。

Jersey 提供的 REST 过滤器和拦截器为开发者提供了很贴心的切面扩展点，开发者无须像在 Spring 中为了针对某个类的方法进行 AOP 扩展，而写配置文件。在 Jersey 中，只要实现相应扩展点的接口，即可实现 REST 请求流程中特定事件点的拦截、扩展，其他工作由底层的 HK2 帮我们做。典型的应用包括请求和响应的过滤和读写拦截。

3.2 Providers 详解

javax.ws.rs.ext.Providers 是 JAX-RS2 定义的一种辅助接口，其实现类用于辅助 REST 框架完成过滤和读写拦截等功能。使用注解 @Provider 来标注这些实现类，可以被 JAX-RS2 的运行时自动探测、加载。Provider 实例可以通过 @Context 注解被依赖注入到其他实例中。Providers 接口定义了 4 个方法，分别用来获取 MessageBodyReader、MessageBodyWriter、ExceptionMapper 和 ContextResolver 实例。

3.2.1 实体 Providers

在 3.3 节我们讲述了 Jersey 支持的各种传输格式。Jersey 之所以可以支持那么多表述的类型，即响应实体的传输格式，是因为其底层实体 Providers 具备的对不同格式的处理能力。Jersey 内部提供了非常丰富的 MessageBodyReader 接口和 MessageBodyWriter 接口实现类，用于处理不同格式的表述，比如字节数组、XML、文件和流等，如图 3-1 所示。本节将揭开读写实体的工具类 MessageBodyReader 和 MessageBodyWriter 面纱，并为我们业务所用。

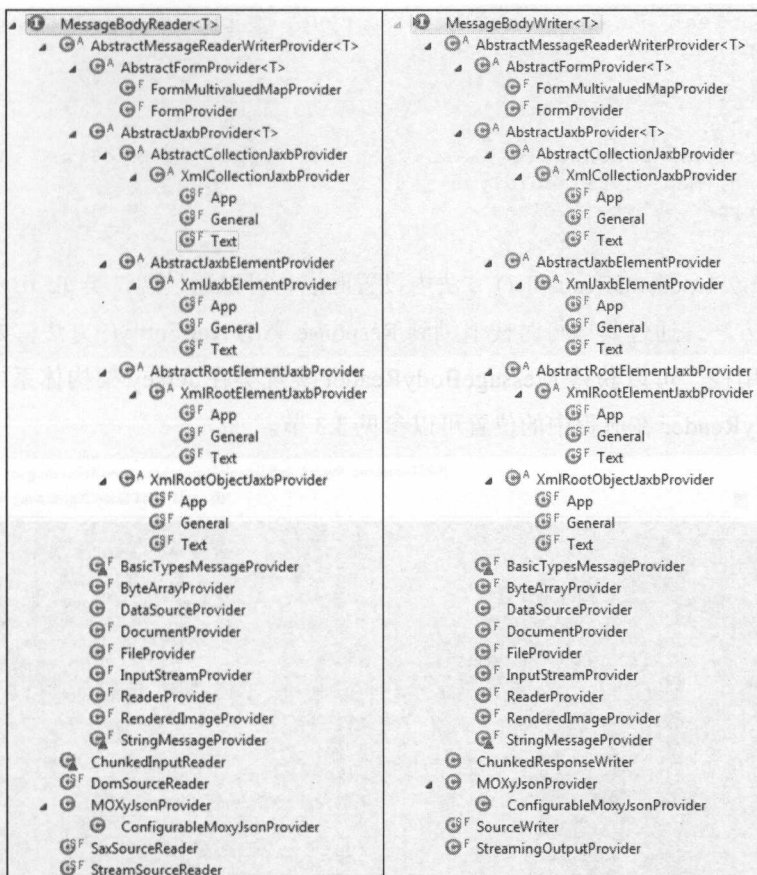


图 3-1 MessageBodyReader 和 MessageBodyWriter 的实现类

1. MessageBodyReader

消息体读处理器接口 `MessageBodyReader<T>` 用于将传输流转换为 Java 类型的对象。`MessageBodyReader` 接口定义了一个泛型，接口的实现类为这个泛型定义一个具体类型，该类型即是该实现类所支持的转换类型。实现类被业务系统启用有两种方式。一是使用注解 `@Provider` 定义实现类，业务系统在启动时自动探测并加载。另一种方式是通过编码注册到 `Application` 类或其子类中，业务系统在启动时，加载 `Application` 类或其子类时一并加载。

(1) isReadable

`MessageBodyReader<T>` 接口定义了两个方法。第一个方法 `isReadable()` 是用来判断实现类是否支持将当前请求的数据类型反序列化。以读取字节数组实体实现类 `ByteArrayProvider` 为例其覆盖方法会判断当前类型是否是字节数组类型，示例代码如下所示。

```
//isReadable 接口方法;
```

```
public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations, MediaType mediaType);
```

//isReadable 方法的实现:

@Override

```
public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations, MediaType mediaType) {
    return type == byte[].class;
}
```

在 `ByteArrayProvider.isReadable()` 方法内设置断点, 可以通过测试类 `TestByteArrayReader` 的 `testReader()` 方法, 通过其运行时栈追溯到 `Response` 类的 `readEntity()` 方法, 如图 3-2 所示。从这期间的调用栈, 可以获得 `MessageBodyReader` 实现类在 Jersey 架构体系中的位置。关于 `MessageBodyReader` 在流程中的位置可以参见 3.3 节。

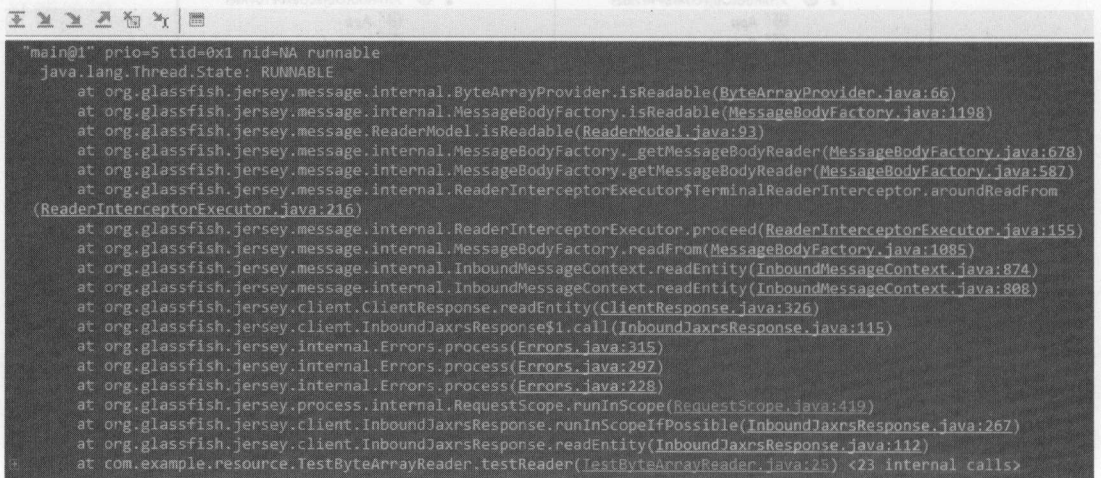


图 3-2 isReadable() 示例

(2) readFrom

`MessageBodyReader<T>` 接口定义的第二个方法 `readFrom()` 是用于处理反序列化, 是处理读取流并转换为 Java 类型对象的核心方法。该方法定义如下。

```
public T readFrom(Class<T> type, Type genericType, Annotation[] annotations,
    MediaType mediaType, MultivaluedMap<String, String> httpHeaders, InputStream
    entityStream) throws java.io.IOException, javax.ws.rs.WebApplicationException;
```

继续读取字节数组实体实现类 `ByteArrayProvider` 的研究, 该方法的覆写包含两个内容。第一是将实体输入流写入字节数组输出流, 第二是将该流以字节数组的形式返回。运行单元测试类 `TestByteArrayReader`, 并在 `ByteArrayProvider` 类中设置断点, 可以监控到具体的行为, 如图 3-3 所示。

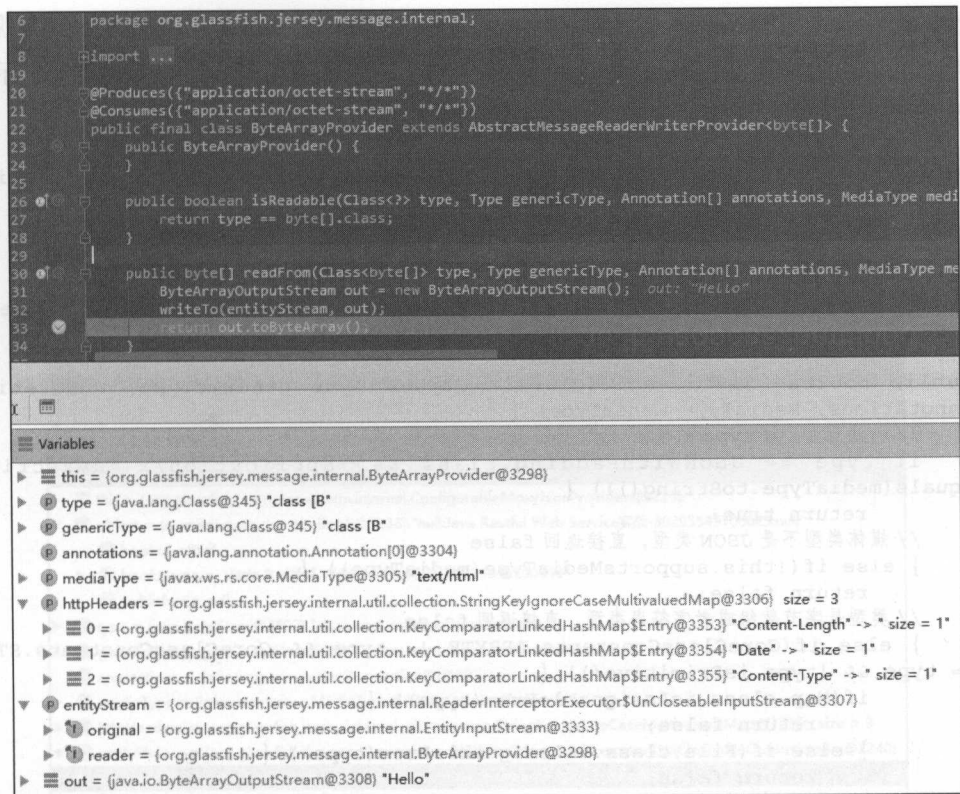


图 3-3 readFrom() 示例

2. MessageBodyWriter

消息体写处理器接口 `MessageBodyWriter<T>` 用于将 Java 类型的对象转换为流，它是序列化的过程和 `MessageBodyReader<T>` 接口实现的反序列化的逆过程。两个接口的设计原理是相同的。对应地，`MessageBodyWriter<T>` 定义了两个方法 `isWriteable()` 和 `writeTo()`。其实，解析一种传输类型的 `Provider` 类通常会同时实现 `MessageBodyReader` 和 `MessageBodyWriter` 这两个接口，比如上面提及的 `ByteArrayProvider` 类。

为了更全面地掌握实体 `Provider` 实现类，并序列化过程的分析，我们选用较为复杂的 `MOXyJsonProvider` 类做例子。`MOXyJsonProvider` 类是第 2 章讲述的 4 种 JSON 支持技术中 `MOXy` 的支持类，来自 `EclipseLink` 项目 (`org.eclipse.persistence.moxy-2.6.0.jar`)。`MOXyJsonProvider` 类实现了上述的两个接口，并定义了生产和消费的媒体类型。

(1) isWriteable

`isWriteable()` 方法用于检测实现类是否支持序列化当前请求的类型，如果不可写，Jersey 框架会放弃使用这个实现类来处理当前的表述。在分析 `ByteArrayProvider` 类的可读

实现时,该方法只使用到了第一个参数 `type`,而在 `MOXyJsonProvider` 类的可写检测的覆写中,同时用到了最后一个参数 `mediaType` 来校验请求的媒体类型是否是 JSON 类型,示例代码如下所示。

```
public boolean isWriteable(Class<?> type, Type genericType, Annotation[]
annotations, MediaType mediaType);
package org.eclipse.persistence.jaxb.rs
@Produces({"application/json", "*/*", "application/x-javascript"})
@Consumes({"application/json", "*/*"})
public class MOXyJsonProvider implements MessageBodyReader<Object>,
MessageBodyWriter<Object> {
    ...
    public boolean isWriteable(Class<?> type, Type genericType, Annotation[]
annotations, MediaType mediaType) {
        // 关注点 1: 对 type 的校验
        if (type == JSONWithPadding.class && "application/x-javascript".
equals(mediaType.toString())) {
            return true;
        }
        // 媒体类型不是 JSON 类型, 直接返回 false
        } else if (!this.supportsMediaType(mediaType)) {
            return false;
        }
        // 类型是字节数组或者字符串类型, 直接返回 false
        } else if (CoreClassConstants.APBYTE != type && CoreClassConstants.STRING
!= type && !type.isPrimitive()) {
            if (Map.class.isAssignableFrom(type)) {
                return false;
            }
            } else if (File.class.isAssignableFrom(type)) {
                return false;
            }
        }
        ...
    } else {
        return false;
    }
}
```

在这段代码中,在关注点 1 处, `MOXyJsonProvider` 类对第一个参数 `type` 类型的校验条件繁多,从字节数组到 `Collection` 类的子类,但凡匹配就返回 `false` 了,就是说 `MOXyJsonProvider` 类不负责对这些类型进行序列化操作。

(2) writeTo

`writeTo()` 方法是请求对象写入流的序列化过程。`MOXyJsonProvider` 类的 `writeTo()` 方法的覆写内容有 30 多行,基本都是对 `Marshaller` 实例的配置,关键的一行是执行 `Marshaller` 实例的 `marshal()` 一行,示例代码如下。

```
//writeTo 接口方法:
public void writeTo(T t, Class<?> type, Type genericType, Annotation[]
annotations, MediaType mediaType, MultivaluedMap<String, Object> httpHeaders, OutputStream
entityStream) throws java.io.IOException, javax.ws.rs.WebApplicationException;

//MOXyJsonProvider 类的 writeTo 方法:
...
marshaller.marshal(object, entityStream);
```


其对应的测试栈信息如图 3-4 所示。

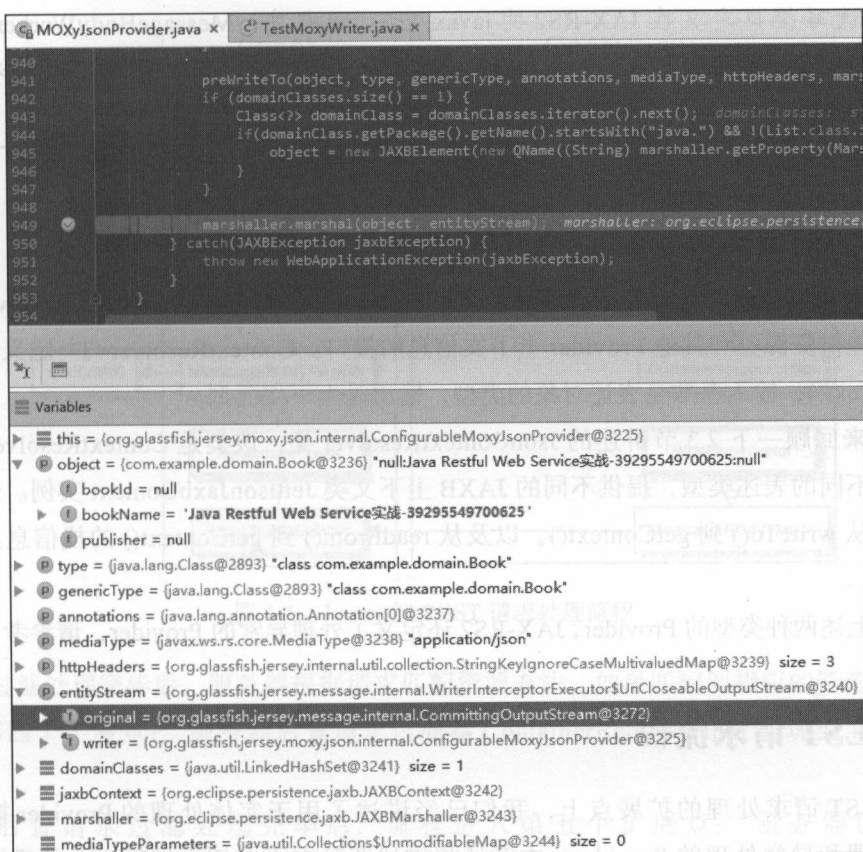


图 3-4 writeTo() 方法示例

在图 3-4 中，从测试类 TestMoxyWriter 的 testMoxyWriter() 方法的栈信息中，可以看到 Book 实例被序列化到 CommittingOutputStream 流实例中，实现了写入的过程。

3. MessageBodyWorkers

从图 3-1 的 MessageBodyReader 和 MessageBodyWriter 的实现类示意图中可以看出，实体读写接口的实现类非常多，编写选择哪个实现类作为当前请求的读写处理器的算法是非常繁重的工作，MessageBodyWorkers 接口旨在抽象这一遴选工作，其实现类可以通过 @Context 依赖注入到使用 MessageBodyWorkers 的类中。MessageBodyFactory 是 MessageBodyWorkers 接口的实现类。在上述的 isReadable() 示例栈中，可以发现它的身影。

阅读指南

实体读写接口定义在 JAX-RS2 的 `javax.ws.rs.ext` 包中, `MessageBodyWorkers` 定义在 Jersey 核心包 `Jersey-Common` 的 `org.glassfish.jersey.message` 包中, 前者是规范定义的接口, 后者是参考实现定义的接口。不要混淆。

3.2.2 上下文 Providers

除了处理实体的 `Provider`, 处理上下文的 `Provider` 也非常重要。`ContextResolver<T>` 接口是用于提供资源类和其他 `Provider` 上下文信息的接口。`ContextResolver<T>` 定义了一个方法 `getContext()`, 输入参数是表述对象的类型, 输出是上下文泛型。

我们来回顾一下 2.3 节讲述的 `JsonContextResolver` 类, 该类是 `ContextResolver` 的实现类, 根据不同的表述类型, 提供不同的 JAXB 上下文类 `JettisonJaxbContext` 实例。通过断点可以跟踪从 `writeTo()` 到 `getContext()`, 以及从 `readFrom()` 到 `getContext()` 的栈信息。这里不再冗述。

除了上述两种类型的 `Provider`, JAX-RS2 还定义了处理异常的 `Provider`, 请参考 2.5.2 节。

3.3 REST 请求流程

从 REST 请求处理的扩展点上, 我们已经讲述了用于实体处理的 `Provider` 接口以及上下文处理和异常处理的 `Provider`。本章还将讲述两种在面向切面编程中非常重要的特殊 `Provider`: 过滤器 (3.4 节) 和拦截器 (3.5 节)。在进入这个主题之前, 我们需要对 REST 请求处理的流程这条线有明确的认识, 如图 3-5 所示, 这样以来, 才会知道这些点都处于流程的什么位置。只有这样才能清楚地实现对扩展点的开发和调试。

在图 3-5 中, 请求流程中存在 3 种角色, 分别是用户、REST 客户端和 REST 服务器。请求始于请求的发送, 止于调用 `Response` 类的 `readEntity()` 方法, 获取响应实体。

1) 用户提交请求数据, 客户端接收请求, 进入第一个扩展点: “客户端请求过滤器 `ClientRequestFilter` 实现类” 的 `filter()` 方法。

2) 请求过滤处理完毕后, 流程进入第二个扩展点: “客户端写拦截器 `WriterInterceptor` 实现类” 的 `aroundWriteTo()` 方法, 实现对客户端序列化操作的拦截。

3) “客户端消息体写处理器 `MessageBodyWriter`” 执行序列化, 流程从客户端过渡到服务器端。

4) 服务器接收请求, 流程进入第三个扩展点: “服务器前置请求过滤器 `Container-`

RequestFilter 实现类”的 filter() 方法。

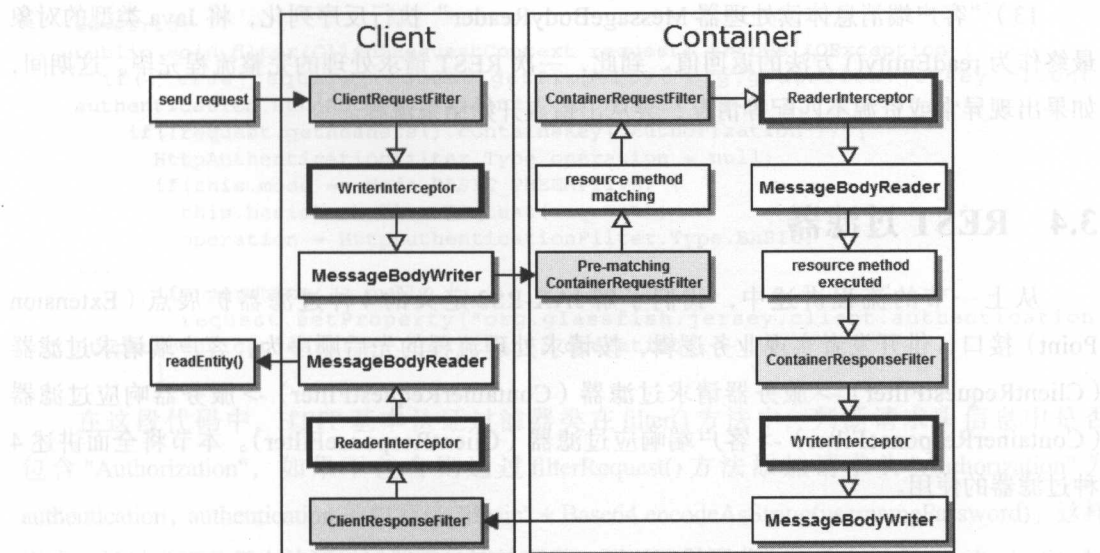


图 3-5 Jersey 的 REST 请求处理流程

5) 过滤处理完毕后，服务器根据请求匹配资源方法，如果匹配到相应的资源方法，流程进入第四个扩展点：“服务器后置请求过滤器 ContainerRequestFilter 实现类”的 filter() 方法。

6) 后置请求过滤处理完毕后，流程进入第五个扩展点：“服务器读拦截器 ReaderInterceptor 实现类”的 aroundReadFrom() 方法，拦截服务器端反序列化操作。

7) “服务器消息体读处理器 MessageBodyReader”完成对客户端数据流的反序列化。服务器执行匹配的资源方法。

8) REST 请求资源的处理完毕后，流程进入第六个扩展点：“服务器响应过滤器 ContainerResponseFilter 实现类”的 filter() 方法。

9) 过滤处理完毕后，流程进入第七个扩展点：“服务器写拦截器 WriterInterceptor 实现类”的 aroundWriteTo() 方法，对服务器端序列化到客户端这个操作的拦截。

10) “服务器消息体写处理器 MessageBodyWriter”执行序列化，流程返回到客户端一侧。

11) 客户端接收响应，流程进入第八个扩展点：“客户端响应过滤器 ClientResponseFilter 实现类”的 filter() 方法。

12) 过滤处理完毕后，客户端响应实例 response 返回到用户一侧，用户执行 response.readEntity() 流程进入第九个扩展点：“客户端读拦截器 ReaderInterceptor 实现类”的

aroundReadFrom() 方法，对客户端反序列化进行拦截。

13) “客户端消息体读处理器 MessageBodyReader” 执行反序列化，将 Java 类型的对象最终作为 readEntity() 方法的返回值。到此，一次 REST 请求处理的完整流程完毕。这期间，如果出现异常或资源不匹配等情况，会从出错点开始结束流程。

3.4 REST 过滤器

从上一节的流程讲述中，我们了解 JAX-RS2 定义的 4 种过滤器扩展点 (Extension Point) 接口，供开发者实现业务逻辑，按请求处理流程的先后顺序为：客户端请求过滤器 (ClientRequestFilter) -> 服务器请求过滤器 (ContainerRequestFilter) -> 服务器响应过滤器 (ContainerResponseFilter) -> 客户端响应过滤器 (ClientResponseFilter)。本节将全面讲述 4 种过滤器的使用。

3.4.1 ClientRequestFilter

客户端请求过滤器 (ClientRequestFilter) 定义的过滤方法 filter() 包含一个输入参数，是客户端请求的上下文类 ClientRequestContext。从该上下文中可以获取请求信息，典型的示例包括获取请求方法 context.getMethod()，获取请求资源地址 context.getUri() 和获取请求头信息 context.getHeaders() 等。过滤器的实现类中可以利用这些信息，覆写该方法以实现该类特有的过滤功能。ClientRequestFilter 接口的实现类如图 3-6 所示。

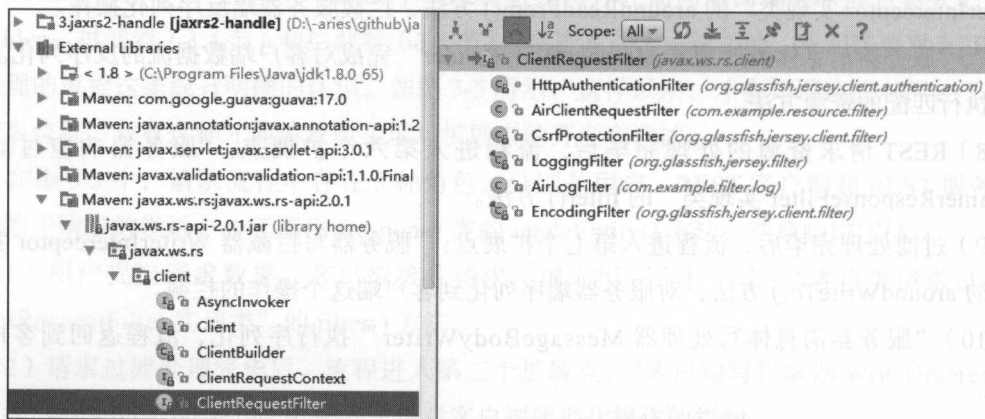


图 3-6 ClientRequestFilter 接口的实现类

图 3-6 展示了 ClientRequestFilter 接口的实现类，包括 Jersey 内部提供的实现类和本书示例代码中的实现类。我们选择 HTTP 认证过滤器类 HttpAuthenticationFilter 作为例子，

来感受上面的讲述, (HTTP 基本认证的内容请参考 10.1 节。) 示例代码如下所示。

```
@Override
public void filter(ClientRequestContext request) throws IOException {
    if(!"true".equals(request.getProperty("org.glassfish.jersey.client.
authentication.HttpAuthenticationFilter.reused"))) {
        if(!request.getHeaders().containsKey("Authorization")) {
            HttpAuthenticationFilter.Type operation = null;
            if(this.mode == Mode.BASIC_PREEMPTIVE) {
                this.basicAuth.filterRequest(request);
                operation = HttpAuthenticationFilter.Type.BASIC;
            }
            if(operation != null) {
                request.setProperty("org.glassfish.jersey.client.authentication.
HttpAuthenticationFilter.operation", operation);
            }
        }
    }
}
```

在这段代码中, HTTP 基本认证过滤器类在 filter() 方法中, 判断请求头信息中是否包含 "Authorization", 如果不包含则通过 filterRequest() 方法添加请求头 "Authorization" 为 authentication, authentication 的内容是 "Basic" + Base64.encodeAsString(usernamePassword)。这样以来, 经过 HTTP 基本认证过滤器类过滤处理后, 可以确保请求头信息中包含 "Authorization"。

3.4.2 ContainerRequestFilter

针对过滤切面, 服务器请求过滤器接口 ContainerRequestFilter 的实现类可以定义为预处理和后处理。默认情况下, 采用后处理方式。即先执行容器接收请求操作, 当服务器接收并处理请求后, 流程才进入过滤器实现类的 filter() 方法。而预处理是在服务器处理接收到的请求之前就执行过滤。如果希望实现一个预处理的过滤器实现类, 需要在类名上定义注解 @PreMatching。

服务器请求过滤器定义的过滤方法 filter() 包含一个输入参数, 即容器请求上下文类 ContainerRequestContext。ContainerRequestFilter 接口的实现类, 如图 3-7 所示。

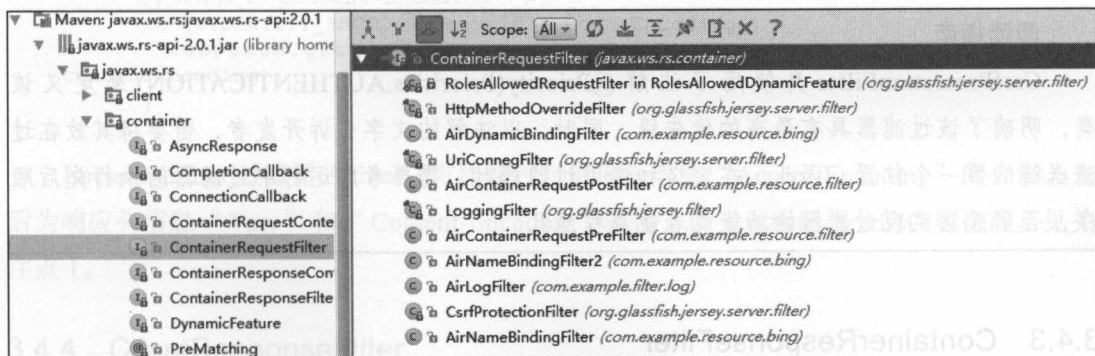


图 3-7 ContainerRequestFilter 接口的实现类

图 3-7 展示了 ContainerRequestFilter 接口的实现类，我们以 CsrfProtectionFilter 为例来说明，示例代码如下所示。

```
package org.glassfish.jersey.server.filter;

@Priority(Priorities.AUTHENTICATION) //post-matching
public class CsrfProtectionFilter implements ContainerRequestFilter {
    public static final String HEADER_NAME = "X-Requested-By";
    // 关注点 1 忽略方法集合
    private static final Set<String> METHODS_TO_IGNORE;
    static {
        HashSet<String> mti = new HashSet<>();
        mti.add("GET");
        mti.add("OPTIONS");
        mti.add("HEAD");
        METHODS_TO_IGNORE = Collections.unmodifiableSet(mti);
    }

    @Override
    public void filter(ContainerRequestContext rc) throws IOException {
        // 关注点 2 判断方法名称是否符合条件
        if (!METHODS_TO_IGNORE.contains(rc.getMethod()) && !rc.getHeaders().containsKey(HEADER_NAME)) {
            throw new BadRequestException();
        }
    }
}
```

在这段代码中，CsrfProtectionFilter 定义了一个特殊的头信息 "X-Requested-By" 和 CSRF 忽略监控的方法集合，见关注点 1。在过滤器的 filter() 方法中，首先从上下文中获取头信息 rc.getHeaders() 和请求方法信息 rc.getMethod()，然后判断头信息是否包含 "X-Requested-By"，方法信息是否是安全的请求方法，即 "GET"、"OPTIONS" 或 "HEAD"。如果两个条件都不成立，过滤器会抛出一个运行时异常 BadRequestException，见关注点 2。通过 CsrfProtectionFilter 过滤器，可以确保请求是 CSRF 安全的。

阅读指南

CsrfProtectionFilter 类使用了注解 @Priority(Priorities.AUTHENTICATION) 来定义该类，明确了该过滤器具有最高的优先级。同时，以注解的文字告诉开发者，需要将其放在过滤器链的第一个位置。因此，在定义和使用过滤器时，需要考虑运行中过滤器的执行先后顺序，否则无法实现过滤器的功能或者使流程混乱。

3.4.3 ContainerResponseFilter

服务器响应过滤器接口 ContainerResponseFilter 定义的过滤方法 filter() 包含两个输

入参数，一个是容器请求上下文类 `ContainerRequestContext`，另一个是容器响应上下文类 `ContainerResponseContext`。`ContainerResponseFilter` 接口的实现类如图 3-8 所示。

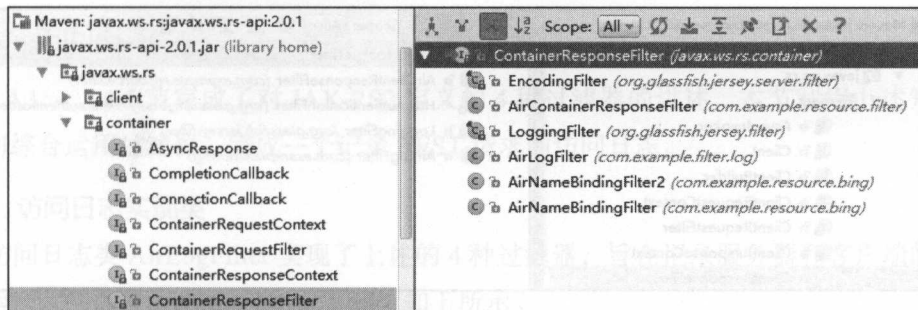


图 3-8 `ContainerResponseFilter` 接口的实现类

图 3-8 展示了 `ContainerResponseFilter` 接口的实现类，我们以 `EncodingFilter` 为例来说明。该过滤器的作用是完成内容协商中编码匹配的工作（内容协商这个知识点请参考 3.6 节），示例代码如下所示。

```
@Priority(Priorities.HEADER_DECORATOR)
public final class EncodingFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext request, ContainerResponseContext
    response) throws IOException {
        ...
        List<String> varyHeader = ((ContainerResponse) response).
        getStringHeaders().get(HttpHeaders.VARY);
        // 关注点 1: Vary 头信息
        if (varyHeader == null || !varyHeader.contains(HttpHeaders.ACCEPT_
        ENCODING)) {
            response.getHeaders().add(HttpHeaders.VARY, HttpHeaders.ACCEPT_
            ENCODING);
        }
        ...
        // 关注点 1: Content-Encoding 头信息
        if (!IDENTITY_ENCODING.equals(contentEncoding)) {
            response.getHeaders().putSingle(HttpHeaders.CONTENT_ENCODING,
            contentEncoding);
        }
    }
}
```

`EncodingFilter` 过滤器的 `filter()` 方法通过对请求头信息“Accept-Encoding”的分析，先后为响应头信息“Vary”和“Content-Encoding”赋值，以实现编码部分的内容协商。见关注点 1。

3.4.4 ClientResponseFilter

客户端响应过滤器（`ClientResponseFilter`）定义的过滤方法 `filter()` 包含两个输入参

数，一个是客户端请求的上下文类 `ClientRequestContext`，另一个是客户端响应的上下文类 `ClientResponseContext`。`ClientResponseFilter` 接口的实现类，如图 3-9 所示。

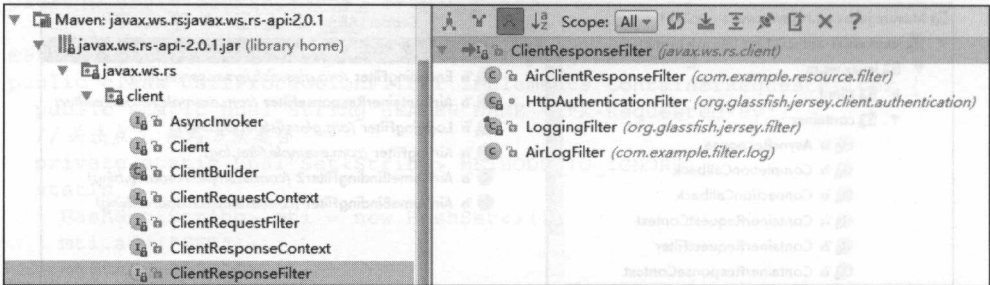


图 3-9 `ClientResponseFilter` 接口的实现类

图 3-9 展示了 `ClientResponseFilter` 接口的实现类，包括 Jersey 内部提供的实现类和本书示例代码中的实现类。我们以 HTTP 摘要认证过滤器类 `HttpAuthenticationFilter` 为例进行演示，(HTTP 摘要认证请参考 10.1 节。) 示例代码如下所示。

```
@Override
public void filter(ClientRequestContext request, ClientResponseContext
response) throws IOException {
    ...
    if(response.getStatus() == Status.UNAUTHORIZED.getStatusCode()) {
        String operation = (String)response.getHeaders().getFirst("WWW-Authenticate");
        if(operation != null) {
            String success = operation.trim().toUpperCase();
            if(success.startsWith("BASIC")) {
                result = HttpAuthenticationFilter.Type.BASIC;
            } else {
                if(!success.startsWith("DIGEST")) {
                    return;
                }
                result = HttpAuthenticationFilter.Type.DIGEST;
            }
        }
        authenticate = true;
    } else {
        authenticate = false;
    }

    if(this.mode != Mode.BASIC_PREEMPTIVE) {
        if(this.mode == Mode.BASIC_NON_PREEMPTIVE) {
            if(authenticate && result == HttpAuthenticationFilter.Type.BASIC) {
                this.basicAuth.filterResponseAndAuthenticate(request, response);
            }
        } else if(this.mode == Mode.DIGEST) {
            if(authenticate && result == HttpAuthenticationFilter.Type.DIGEST) {
                this.digestAuth.filterResponse(request, response);
            }
        }
    }
}
```

```

    }
    ...
}

```

3.4.5 访问日志

3.4.1 ~ 3.4.5 节完成了对 JAX-RS2 定义的 4 种过滤器的讲述，本节利用上述知识，演示如何综合运用过滤器，完成一个记录 REST 请求的访问日志。

1. 访问日志实现类

访问日志类 `AirLogFilter` 实现了上述的 4 种过滤器，旨在记录服务器和客户端的请求和响应运行时的信息。`AirLogFilter` 类定义如下所示。

```

@Override
public class AirLogFilter implements ContainerRequestFilter, ClientRequestFilter,
    ContainerResponseFilter, ClientResponseFilter {

```

`AirLogFilter` 为每一种过滤器接口定义的 `filter()` 方法提供了实现。在客户端请求过滤中，输出请求资源地址信息和请求头信息；在容器请求过滤中，输出请求方法、请求资源地址信息和请求头信息；在容器响应过滤中，输出 HTTP 状态码和请求头信息；在客户端响应过滤中，输出 HTTP 状态码和请求头信息。4 个阶段的 `filter()` 示例代码如下所示。

```

@Override
public void filter(ClientRequestContext context) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    // 关注点 1: 获取请求方法和地址
    printRequestLine(CLIENT_REQUEST, b, id, context.getMethod(), context.getUri());
    // 关注点 2: 获取请求头信息
    printPrefixedHeaders(CLIENT_REQUEST, b, id, HeadersFactory.asStringHeaders
        (context.getHeaders()));
    LOGGER.info(b.toString());
}

```

在这段代码中，`AirLogFilter` 类实现了客户端请求过滤。从客户端请求上下文实例中，可以获取到请求方法和请求地址信息，见关注点 1。同样，头信息也可以从中获取，见关注点 2。

```

@Override
public void filter(ClientRequestContext requestContext, ClientResponseContext
    responseContext) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    // 关注点 1: 获取响应状态
    printResponseLine(CLIENT_RESPONSE, b, id, responseContext.getStatus());
    // 关注点 2: 获取响应头信息
    printPrefixedHeaders(CLIENT_RESPONSE, b, id, responseContext.getHeaders());
    LOGGER.info(b.toString());
}

```

在这段代码中，AirLogFilter 类实现了客户端响应过滤。从客户端响应上下文实例中，可以获取到响应状态信息和响应头信息。分别见关注点 1 和关注点 2。

```

` `` ` java
@Override
public void filter(ContainerRequestContext context) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    // 关注点 1: 获取容器请求方法和请求地址信息
    printRequestLine(SERVER_REQUEST, b, id, context.getMethod(), context.
getUriInfo().getRequestUri());
    // 关注点 2: 获取请求头信息
    printPrefixedHeaders(SERVER_REQUEST, b, id, context.getHeaders());
    LOGGER.info(b.toString());
}

```

在这段代码中，AirLogFilter 类实现了容器请求过滤。从容器请求上下文实例中，可以获取到请求方法和请求资源地址信息，见关注点 1。同样，可以从中获取请求头信息，见关注点 2。

```

@Override
public void filter(ContainerRequestContext requestContext,
ContainerResponseContext responseContext) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    // 关注点 1: 获取容器响应状态
    printResponseLine(SERVER_RESPONSE, b, id, responseContext.getStatus());
    // 关注点 2: 获取容器响应头信息
    printPrefixedHeaders(SERVER_RESPONSE, b, id, HeadersFactory.
asStringHeaders(responseContext.getHeaders()));
    LOGGER.info(b.toString());
}

```

在这段代码中，AirLogFilter 类实现了容器响应过滤。从容器响应上下文实例中，可以获取到容器的响应状态信息和响应头信息。分别见关注点 1 和关注点 2。

2. 单元测试类

访问日志的单元测试示例如下所示。

```

public class TIResourceJtfTest extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        return config.register(com.example.filter.log.AirLogFilter.class);
    }
    @Override
    protected void configureClient(ClientConfig config) {
        config.register(new AirLogFilter());
    }
}

```

在这段代码中，为了使访问日志类生效，需要测试类 TIResourceJtfTest 在 Jersey 测试

框架的服务器端和客户端，分别注册服务日志类 AirLogFilter。

单元测试的结果如下所示，在 4 种过滤器中分别打印了该阶段的日志信息。

```
main - 1 * AirLog - Request received on thread main
1 / GET http://localhost:9998/books/1
1 / Accept: application/json
Grizzly-worker(1) - 2 * AirLog - Request received on thread Grizzly-worker(1)
2 > GET http://localhost:9998/books/1
2 > accept: application/json
...
Grizzly-worker(1) - 3 * AirLog - Response received on thread Grizzly-worker(1)
3 < 200
3 < Content-Type: application/json
main - 4 * AirLog - Response received on thread main
4 \ 200
4 \ Content-Type: application/json
...
```

3.5 REST 拦截器

拦截器和过滤器的相同点是都是一种在请求—响应模型中，用做切面处理的 Provider。两者的不同除了功能性上的差异（一个用于过滤消息，一个用于拦截处理）之外，形式上也不同。拦截器通常读写成对，而且没有服务器端和客户端的区分。Jersey 提供的拦截器类，如图 3-10 所示。

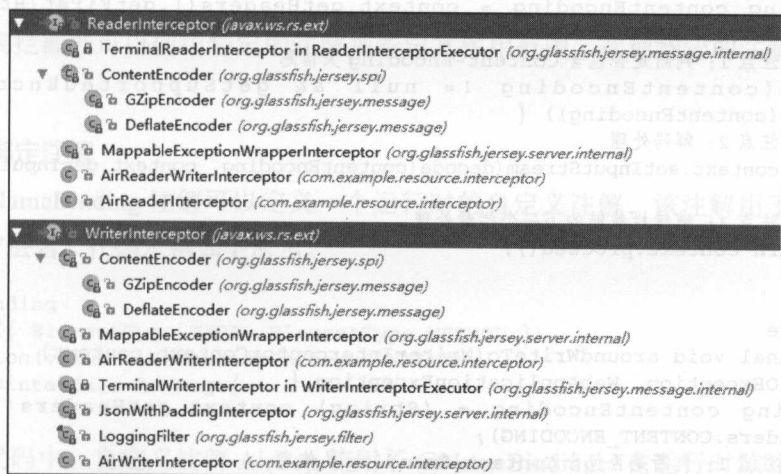


图 3-10 读写拦截器的实现类

在图 3-10 中，Jersey 内部实现了几个典型应用的拦截器，它们是成对出现的。比如 GZipEncoder 同时实现了读写拦截器，以实现使用 GZip 压缩格式压缩消息体的功能。

1. ReaderInterceptor

读拦截器接口 `ReaderInterceptor` 定义的拦截方法是 `aroundReadFrom()`，该方法包含一个输入参数，即读拦截器的上下文接口 `ReaderInterceptorContext`，从中可以获取头信息、输入流以及父接口 `InterceptorContext` 提供的媒体类型等上下文信息。接口方法示例如下。

```
public Object aroundReadFrom(ReaderInterceptorContext context) throws java.io.IOException, javax.ws.rs.WebApplicationException;
```

2. WriterInterceptor

写拦截器接口 `WriterInterceptor` 定义的拦截方法是 `aroundWriteTo()`，该方法包含一个输入参数，写拦截器上下文接口 `WriterInterceptorContext`，从中可以获取头信息、输出流以及父接口 `InterceptorContext` 提供的媒体类型等上下文信息。接口方法示例如下所示。

```
void aroundWriteTo(WriterInterceptorContext context) throws java.io.IOException, javax.ws.rs.WebApplicationException;
```

3. 编解码约束拦截器

编解码约束拦截器类 `ContentEncoder` 是一个位于 `org.glassfish.jersey.spi` 包中的拦截器，SPI 包下的工具是可插拔的。`ContentEncoder` 拦截器用于约束序列化和反序列化的过程中，编解码的内容协商，示例代码如下所示。

```
@Override
public final Object aroundReadFrom(ReaderInterceptorContext context)
throws IOException, WebApplicationException {
    String contentEncoding = context.getHeaders().getFirst(HttpHeaders.
CONTENT_ENCODING);
    // 关注点 1: 判断是否包含 Content-Encoding 头信息
    if (contentEncoding != null && getSupportedEncodings().
contains(contentEncoding)) {
        // 关注点 2: 解码处理
        context.setInputStream(decode(contentEncoding, context.getInputStream()));
    }
    // 关注点 3: 继续拦截链的下一个拦截处理
    return context.proceed();
}

@Override
public final void aroundWriteTo(WriterInterceptorContext context)
throws IOException, WebApplicationException {
    String contentEncoding = (String) context.getHeaders().getFirst(
(HttpHeaders.CONTENT_ENCODING);
    // 关注点 1: 判断是否包含 Content-Encoding 头信息
    if (contentEncoding != null && getSupportedEncodings().contains(
(contentEncoding)) {
        // 关注点 2: 编码处理
        context.setOutputStream(encode(contentEncoding, context.getOutputStream()));
    }
    // 关注点 3: 继续拦截链的下一个拦截处理
```

```

    context.proceed();
}

```

在这段代码中，分别给出了 ContentEncoder 拦截器的读、写拦截处理，只有当头信息包含“Content-Encoding”信息，编解码才被执行，见关注点 1。读取阶段进行解码，写入阶段进行编码，见关注点 2。上下文的 proceed() 方法用于执行拦截器链的下一个拦截器，见关注点 3。

3.6 绑定机制

在我们了解了面向切面的 Providers 的功能后，需要掌握它们是如何加载的，以及其作用域。这些容器级别的 Providers，通常使用编码的方式注册到 Application 中，但这不是唯一的办法。本节将详细讨论 Providers 的绑定机制。

默认情况下，过滤器和拦截器都是全局绑定的。也就是说，如下之一的过滤器或拦截器是全局有效的。

- 通过手动注册到 Application 或者 Configuration;

- 注解为 @Provider，被自动探测。

下面介绍其他的绑定机制。

3.6.1 名称绑定

过滤器或拦截器可以使用特定的注解来指定其作用范围，这种特定的注解被称为名称绑定。

1. 名称绑定注解

使用 @NameBinding 注解可以定义一个运行时的自定义注解，该注解用于定义类级别名称和类的方法名，代码示例如下所示。

```

@NameBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface AirLog {
}

```

在这段代码中，自定义注解 AirLog 使用了 @NameBinding，在运行时该注解将被解析为一个名称绑定的注解。

2. 绑定 Provider

在定义了 @AirLog 注解后，既可以在 Provider 中使用该注解，示例代码如下所示。

```

// 关注点 1: 使用自定义注解 @AirLog
@AirLog
@Priority(Priorities.USER)
public class AirNameBindingFilter implements ContainerRequestFilter,
ContainerResponseFilter {
    private static final Logger LOGGER = Logger.getLogger(AirNameBindingFilter.class);
    public AirNameBindingFilter() {
        LOGGER.info("Air-NameBinding-Filter initialized");
    }
    @Override
    // 关注点 2: filter 实现访问日志
    public void filter(final ContainerRequestContext containerRequest) throws
IOException {
        LOGGER.debug("Air-NameBinding-ContainerRequestFilter invoked:" +
containerRequest.getMethod());
        LOGGER.debug(containerRequest.getUriInfo().getRequestUri());
    }
    @Override
    // 关注点 3: filter 实现访问日志
    public void filter(ContainerRequestContext containerRequest,
ContainerResponseContext responseContext) throws IOException {
        LOGGER.debug("Air-NameBinding-ContainerResponseFilter invoked:" +
containerRequest.getMethod());
        LOGGER.debug("status=" + responseContext.getStatus());
    }
}

```

在这段代码中，过滤器类 `AirNameBindingFilter` 使用了自定义注解 `@AirLog`，这样 `AirNameBindingFilter` 类就实现了名称绑定，见关注点 1。该类实现了容器的请求和响应过滤器接口，功能是记录访问日志，见关注点 2。

3. 绑定方法

接下来，我们在资源方法级别使用自定义注解 `@AirLog`，来实现在资源类的指定方法上启用 `AirNameBindingFilter` 过滤器，示例代码如下所示。

```

@Path("books")
public class BookResource {
    // 关注点 1: 绑定方法
    @AirLog
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Books getBooks() {
        ...
        return books;
    }
    ...
}

```

在这段代码中，资源类 `BookResource` 包含多个方法，我们只在 `getBooks()` 方法上使用了解析 `@AirLog`，而其他方法并没有绑定，见关注点 1。

4. 单元测试类

接下来，通过单元测试来校验名称绑定的设计和实现是否正确，示例代码如下所示。

```
public class TestNamingBinding extends JerseyTest {
    @Override
    protected Application configure() {
        // 关注点 1: AirAopConfig 内部注册了 AirNameBindingFilter
        return new AirAopConfig();
    }
    @Test
    // 关注点 2: 测试 getBookByPath() 方法
    public void testPathGetJSON() {
        final WebTarget pathTarget = target(BASE_URI).path("1");
        final Invocation.Builder invocationBuilder = pathTarget.request(
            MediaType.APPLICATION_JSON_TYPE);
        final Book result = invocationBuilder.get(Book.class);
        Assert.assertNotNull(result.getBookId());
    }
    @Test
    // 关注点 3: 测试 getBooks() 方法
    public void testGetAll() {
        final Invocation.Builder invocationBuilder = target(BASE_URI).request();
        final Books result = invocationBuilder.get(Books.class);
        Assert.assertNotNull(result.getBookList());
    }
}
```

在这段代码中，测试类 `TestNamingBinding` 通过 `AirAopConfig` 注册了 `AirNameBindingFilter` 过滤器，见关注点 1。该类包含两个测试方法，分别是测试资源类 `BookResource` 的 `getBooks()` 和 `getBookByPath()` 两个方法，见关注点 2 和关注点 3。

我们可以从终端打印的信息来检验名称绑定的运行结果，示例如下。

```
Air-NameBinding-ContainerRequestFilter invoked:GET
http://localhost:9998/books/
Air-NameBinding-ContainerResponseFilter invoked:GET
status=200
```

从上述测试结果中可以看到，只有在 `testGetAll()` 方法输出的日志中输出了 `AirNameBindingFilter` 类中定义的日志信息。这和预期的“只有使用注解 `@AirLog` 定义的方法，才会在请求流程中启用相应的 `Provider`”一致。

3.6.2 动态绑定

名称绑定需要通过自定义的注解名称来绑定 `Provider` 和扩展点方法或者类，相比而言，动态绑定无须新增注解，而是使用编码的方式，实现动态特征接口 `javax.ws.rs.container.DynamicFeature`，定义扩展点方法的名称、请求方法类型等匹配信息。在运行期，一旦 `Provider` 匹配当前处理类或方法，面向切面的 `Provider` 方法即被触发。

1. 定义绑定 Provider

AirDynamicFeature 类实现了 DynamicFeature 接口，示例代码如下。

```
public class AirDynamicFeature implements DynamicFeature {
    @Override
    public void configure(final ResourceInfo resourceInfo, final FeatureContext context) {
        boolean classMatched = BookResource.class.isAssignableFrom(resourceInfo.
            getResourceClass());
        boolean methodNameMatched = resourceInfo.getResourceMethod().getName().
            contains("getBookBy");
        boolean methodTypeMatched = resourceInfo.getResourceMethod().
            isAnnotationPresent(POST.class);
        // 关注点 1: 匹配成功才注册 AirDynamicBindingFilter
        if (classMatched && (methodNameMatched || methodTypeMatched)) {
            context.register(AirDynamicBindingFilter.class);
        }
    }
}

public class AirDynamicBindingFilter implements ContainerRequestFilter {
    @Override
    public void filter(final ContainerRequestContext requestContext) throws IOException {
        AirDynamicBindingFilter.LOGGER.debug("Air-Dynamic-Binding-Filter invoked");
    }
}
```

在这段代码中，在 AirDynamicFeature 的配置方法中，启用了如下匹配规则。

- 1) 类匹配：对 BookResource 类及其子类的匹配。
- 2) 方法名称匹配：方法名包含 getBookBy() 的匹配。
- 3) 请求方法类型匹配：与 POST 方法的匹配。

只有当匹配成功时，才会注册 AirDynamicBindingFilter。对于 Provider 的实现类，并没有特殊的要求。

2. 单元测试类

测试类 TestDynamicBinding 注册了动态绑定特征实现类 AirDynamicFeature，示例代码如下所示。

```
java
public class TestDynamicBinding extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        config.register(AirDynamicFeature.class);
        return config;
    }

    @Test
    public void testPost() {
        final Book newBook = new Book("Java Restful Web Service 使用指南 -" + System.
            nanoTime());
    }
}
```



```

final Entity<Book> bookEntity = Entity.entity(newBook, MediaType.
APPLICATION_JSON_TYPE);
final Book savedBook = target(BASE_URI).request(MediaType.APPLICATION_JSON_
TYPE).post(bookEntity, Book.class);
Assert.assertNotNull(savedBook.getBookId());
}
}

```

运行测试方法，AirDynamicBindingFilter 的日志信息如预期输出，示例代码如下所示。

```

Air-Dynamic-Binding-Filter initialized
Air-Dynamic-Binding-Filter invoked

```

3.7 优先级

不同扩展点的 Provider 在请求处理流程中的顺序在 3.3 节已经阐述，对于同一个扩展点的多个 Provider 的执行的先后顺序是靠优先级排序的。优先级的定义使用注解 @Priority，优先级的值是一个整型值，常量定义在 javax.ws.rs.Priorities 类中。对于 ContainerRequest、PreMatchContainerRequest、ClientRequest 和读写拦截器该数值采用升序策略，即数值越小优先级越高；对于 ContainerResponse 和 ClientResponse 该数值采用降序策略，即数越大优先级越高，示例代码如下所示。

```

@Priority(Priorities.USER)
public class AirNameBindingFilter
@Priority(Priorities.USER + 1)
public class AirNameBindingFilter2

```

在这段代码中，AirNameBindingFilter2 的 Priority 数值大于 AirNameBindingFilter。在请求过程中，AirNameBindingFilter 优先执行；在响应过程中，AirNameBindingFilter2 被优先执行。

单元测试代码如下所示。

```

public class TestPriority extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        config.register(AirNameBindingFilter.class);
        config.register(AirNameBindingFilter2.class);
        return config;
    }
    @Test
    public void testGetAll() {
        final Invocation.Builder invocationBuilder = target(BASE_URI).request();
        final Books result = invocationBuilder.get(Books.class);
        Assert.assertNotNull(result.getBookList());
    }
}

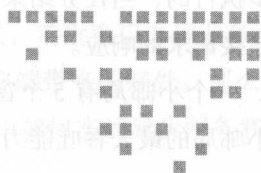
```

单元测试结果如下所示, 处理请求阶段, `AirNameBindingFilter2` 执行在后; 响应阶段 `AirNameBindingFilter2` 执行在先。

```
Air-NameBinding-ContainerRequestFilter invoked:GET
http://localhost:9998/books/
Air-NameBinding-ContainerRequestFilter2 Priority+1 invoked
Air-NameBinding-ContainerResponseFilter2 Priority+1 invoked
Air-NameBinding-ContainerResponseFilter invoked:GET
status=200
```

3.8 本章小结

本章详述了 JAX-RS2 定义的 `Provider` 及其两种特殊类型过滤器和拦截器, 并详细讲述了 JAX-RS2 对 REST 请求的处理过程中各个扩展点对应的 `Provider`。随后讲述了 `Provider` 的绑定机制和优先级机制。



第 4 章

Chapter 4

REST 服务与异步

有时，异步可以为 REST 服务带来更强大的功能、提高服务的性能以及提升用户体验。本章将详述异步机制和异步通信，以及基于 JAX-RS2 标准的实践。首先我们将一起思考，异步机制到底有什么用处，随后详述 JAX-RS2 定义的异步请求处理规范。在异步机制的理论和实践之后，我们分别讲述基于 HTTP1.1 协议和 HTML5 的异步通信。

4.1 为什么使用异步机制

在涉及性能的话题中，我们通常会考虑启用异步机制。那么异步在何时可以提高性能呢？启用异步机制的前提是同步运行时资源存在空置。

接下来，我们从服务器和客户端两个角度来思考引入异步要解决的问题以及如何来解决。

4.1.1 服务器异步机制

服务器端使用异步机制的主要目的是将“处理连接”与“处理请求”解耦。

对于服务器而言，如果处理连接的线程被一个需要较长时间才能处理完毕的任务阻塞，那么服务器处理连接的能力就会下降，而此时服务器的资源很有可能是空闲的。此时，我们考虑将处理连接和处理请求任务解耦，处理连接的线程接收请求后，将其分派给处理请求任务的线程。这样一来，即使任务需要较长时间才能完成，处理连接的线程也无需阻塞等待了，服务器因此可以重用连接线程，从而提供更高的吞吐率。处理请求的线程相对于处理连

接的线程是异步执行的，当任务结束后，服务器会从上下文中找到当前连接，并将处理结果返回，作为该连接请求的响应。

举个例子，一个小邮局有 5 个窗口，如果每个窗口的工作人员都要全程处理顾客的邮寄业务，那么小邮局的最大吞吐能力为 5 个，一旦有一个窗口受理的业务非常耗时，吞吐量就减少一个。

如果每个窗口的接待人员只负责理解顾客业务并将其分类分派给邮局的其他工作人员具体处理，那么邮局中处理业务的人员与顾客之间的沟通不会阻碍窗口接待人员，窗口的吞吐能力会一直保持在 5 个。

我们进一步来看这个场景。处理任务的线程在任务处理过程中是同步阻塞的，如果任务是可分解的，那么我们可以考虑使用多线程来同时分别处理分解后的任务，最后将其合并。这样一来，处理任务的线程在任务处理过程中，就处于异步阻塞状态，其所用时间由分解后任务中最长的子任务处理时间决定。

当受理的业务是要邮寄由多个小包装组成的包裹时，假设会过来好几个工作人员帮忙，一个确认小包装的重量，一个找来了大包装，一个准备好胶带，还有一个帮助确认邮寄地址，很快一个标邮寄地址、重量，并确定邮寄费用的大包装就搞定了。

从这个角度看，异步设计的确可以提高服务器的性能。但是，并非异步就一定能提高性能。因为，异步可以提高性能的前提如下。

- 任务执行的时间要远大于创建异步线程所消耗的时间。

- 服务器资源在同步阻塞情况下是有资源空闲的。

如果小邮局的工作人员只有 6 个人，开 5 个窗口就很难再分派受理业务了，因为窗口里面只剩余一个人，他无法一次处理 5 个业务，顾客只能默默地等待了。

4.1.2 客户端异步机制

对于客户端而言，无论服务器是同步还是异步，如果客户端需要等待请求响应后，才处理其他事情（比如浏览器的行为是渲染 HTML 页面），那么客户端的行为都是同步且阻塞的。

我们还举上面这个例子。小邮局的顾客排起了长队，一个接一个地办理业务，虽然每个窗口都采用了分派的方式来受理邮寄业务，但对于每个队伍而言，大家还得等前一个被受理后才能走近窗口办理业务。

为了让客户端的流程不受服务器端处理过程的阻塞，可以在客户端启用异步机制。在请求发出前，先注册事件通知（使用观察者模式实现的回调机制），请求发出后，流程继续执行而不等待。当响应到达后，客户端处理响应信息，更新状态。

于是，小邮局启用了叫号机制，顾客到了小邮局的第一件事就是按业务拿号，然后无需排队，可以忙一会儿自己手头的事情，待叫号的时候，走近窗口办理业务就好了。

综上所述，我们可以看到异步机制为服务器和客户端带来的好处。那么，我们在 REST 中如何实现异步呢？接下来，我们将详述在 JAX-RS2 中如何实现异步服务和异步请求。

4.2 JAX-RS2 的异步机制

基于 4.1 节的理论，我们来看看 JAX-RS2 是如何支持异步的。

在 Java 领域，Java 语言的并发处理，Java SE 中 5.0 是一个里程碑；而 Java EE 的并发支持，Java EE 7.0 是一个重要的版本。容器级别上有了对并发的支持，客户端等待服务器的响应就可以由一个 Future 实现，感觉上就像 Java SE 开发中，等待同一个 JVM 的另一个线程一样。在 JAX-RS2 的异步实现过程中，线程是由容器管理的，这是 Java EE 7 中 JSR236 规范定义的功能，读者可以参见《Java EE 7 精粹》一书的第 10 章。

了解了 JAX-RS2 异步处理的流程，接下来我们进入实践。

阅读指南

4.2 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/>

4.2.asynchronized。

4.2.1 服务端实现

对于 JAX-RS2 的服务器端，实现异步主要包括两个技术点：一个是资源方法中对 AsyncResponse 的使用，另一个是对异步机制中 CompletionCallback 和 TimeoutHandler 接口的实现。本节我们将通过一个示例来讲述服务器端的异步实现。

假设图书资源要支持全量查询，而这个查询的过程是很耗时的。我们将利用 JAX-RS2 提供的 AsyncResponse，通过一个异步线程来执行查询，在查询完成后，由这个异步线程完成对请求的响应。

1. 异步资源类

我们首先来看这个支持异步的图书资源类 AsyncResource，其资源地址定义为 books，示例代码如下。

```
@Path("books")
```

```
public class AsyncResource {
```



```

private final ExecutorService threadPool = Executors.newFixedThreadPool(10);

@GET
/** 关注点 1 **/
public void getAll(@Suspended final AsyncResponse asyncResponse) {
    configResponse(asyncResponse);
    /** 关注点 2 **/
    threadPool.submit(new BatchRunner(asyncResponse));
}

class BatchRunner implements Runnable {
    private final AsyncResponse response;

    public BatchRunner(AsyncResponse asyncResponse) {
        this.response = asyncResponse;
    }

    @Override
    public void run() {
        try {
            Books books = queryAll();
            /** 关注点 3 **/
            response.resume(books);
        } catch (InterruptedException e) {
            log.error(e);
        }
    }
}

private Books queryAll() throws InterruptedException {
    Books books = new Books();
    for (int i = 0; i < ThreadLocalRandom.current().nextInt(5, 10); i++) {
        Thread.sleep(500);
        Book book = new Book(i + 100001, "Java RESTful Web Services", "华章");
        log.debug(book);
        books.getBookList().add(book);
    }
    return books;
}
}

```

在这段代码中，GET 方法 `getAll()` 用于处理全量查询（通过 `queryAll()` 方法来模拟一个耗时的处理场景）。该方法的参数是异步响应类 `AsyncResponse` 的实例（习惯上，位于第一位的参数是上下文环境变量参数，如果有业务参数，应当置于其后），使用注解 `@Suspended` 来标识，见关注点 1。我们将耗时的查询交由一个异步线程执行，见关注点 2。当查询执行结束，异步响应实例的 `resume()` 方法被调用，请求处理线程被唤醒，返回值将作为 `resume()` 方法的参数响应给客户端，见关注点 3。

除了以上 3 个关注点，我们会发现 `getAll` 包含了 `configResponse` 这样一个方法。该方法是由于定义回调的。接下来我们来看看回调的具体实现。

2. 回调方法

CompletionCallback 是 JAX-RS2 定义的用于处理异步完成的接口。当请求处理完成时，CompletionCallback 实例的 onComplete() 方法会被回调。实现 onComplete 方法，可以监听请求处理完成事件并实现相关业务流程。CompletionCallback 的实现作为 AsyncResource 的 register() 方法的参数来配置，这样配置后，AsyncResource 实例会在 resume() 被调用后执行回调方法 onComplete，示例代码如下。

```
asyncResponse.register(new CompletionCallback() {
    @Override
    public void onComplete(Throwable throwable) {
        if (throwable == null) {
            LOGGER.info("CompletionCallback-onComplete: OK");
        } else {
            LOGGER.info("CompletionCallback-onComplete: ERROR: " + throwable.
                getMessage());
        }
    }
});
```

相应的 Java8 Lamda 形式的实现如下，余下的两个回调我们直接使用 Lamda 形式。

```
asyncResponse.register((CompletionCallback) throwable -> {
    if (throwable == null) {
        log.info("CompletionCallback-onComplete: OK");
    } else {
        log.info("CompletionCallback-onComplete: ERROR: " + throwable.
            getMessage());
    }
});
```

ConnectionCallback 是 JAX-RS2 定义的连接断开的接口，当请求一响应模型的连接断开时，ConnectionCallback 实例的 onDisconnect() 方法会被回调。实现 onDisconnect 方法可以监听连接断开事件并实现相关业务，比如主动唤醒 AsyncResource 实例并设置 HTTP 状态码为 410、客户端请求资源不可用 (Response.Status.GONE) 来完成响应。ConnectionCallback 的实现可以作为 AsyncResource 的 register() 方法的参数来配置，示例代码如下。

```
asyncResponse.register((ConnectionCallback) disconnected -> {
    log.info("ConnectionCallback-onDisconnect");
    //Status.GONE=410
    disconnected.resume(Response.status(Response.Status.GONE).entity(
        "disconnect!").build());
});
```

TimeoutHandler 是 JAX-RS2 定义的超时处理器接口，用于处理异步响应类超时事件。当预期的超时时间到达后，TimeoutHandler 实例的 handleTimeout() 方法会被调用。实现 handleTimeout 方法可以监听超时事件并处理相关业务，比如主动唤醒 AsyncResource 实例，

并设置 HTTP 状态码为 503、服务器端服务不可用 (Response.Status.SERVICE_UNAVAILABLE) 来完成响应。TimeoutHandler 的实现可以作为 AsyncResource 的 setTimeoutHandler() 方法的参数来配置。AsyncResource 的 setTimeout() 方法用于设置超时时间, 默认情况下 AsyncResource 永不超时, 示例代码如下。

```
asyncResponse.setTimeoutHandler(asyncResponse0 -> {
    log.info("TIMEOUT");
    //Status.SERVICE_UNAVAILABLE=503
    asyncResponse0.resume(Response.status(Response.Status.SERVICE_UNAVAILABLE).entity("Operation time out.").build());
});
asyncResponse.setTimeout(TIMEOUT, TimeUnit.SECONDS);
```

4.2.2 客户端实现和测试

相应地, JAX-RX 为客户端提供了用于执行异步请求的 API。开发者使用这套 API 可以轻松地实现对服务器端的异步请求。

1. 异步测试类

首先我们来实现基本的异步请求, 示例代码如下。

```
@Test
public void testAsync() throws InterruptedException, ExecutionException {
    final Invocation.Builder request = target("http://localhost:" + this.port +
        "/books").request();
    /** 关注点 1 **/
    final AsyncInvoker async = request.async();
    final Future<Books> responseFuture = async.get(Books.class);
    long beginTime = System.currentTimeMillis();
    try {
        /** 关注点 2 **/
        Books result = responseFuture.get(AsyncResource.TIMEOUT + 1, TimeUnit.SECONDS);
        log.debug("Testing result size = {}", result.getBookList().size());
    } catch (TimeoutException e) {
        /** 关注点 3 **/
        log.debug("Fail to request asynchronously", e);
    } finally {
        log.debug("Elapsed time = {}", System.currentTimeMillis() - beginTime);
    }
}
```

在这段代码中, 客户端使用 AsyncInvoker 接口的 get() 方法提交异步请求, 见关注点 1。该方法返回 Future 接口的实例, 客户端线程可以以非阻塞的方式处理其他业务流程, 然后调用 Future 的 get() 方法获取服务器处理结果, 见关注点 2。如果在指定的时间内, 服务器没有响应, 将报 TimeoutException 异常, 我们可以在异常捕获中实现超时处理, 见关注点 3。

2. 回调方法

客户端亦可以实现异步调用的回调。在 `AsyncInvoker` 接口的 `get()` 方法中, 定义 `InvocationCallback` 接口的实例, 即可实现对 REST 请求的回调处理, 示例代码如下。

```
@Test
public void testAsyncCallBack() throws InterruptedException,
    ExecutionException {
    final AsyncInvoker async = target("http://localhost:" + this.port + "/"
    books").request().async();
    final Future<Books> responseFuture = async.get(new InvocationCallback<Books>() {
        @Override
        /** 关注点 1 */
        public void completed(Books result) {
            log.debug("On Completed: " + result.getBookList().size());
        }

        @Override
        /** 关注点 2 */
        public void failed(Throwable throwable) {
            log.debug("On Failed: " + throwable.getMessage());
            throwable.printStackTrace();
        }
    });
    log.debug("First response time::" + System.currentTimeMillis());
    try {
        responseFuture.get(AsyncResource.TIMEOUT + 1, TimeUnit.SECONDS);
    } catch (TimeoutException e) {
        log.debug("", e);
    } finally {
        log.debug("Second response time::" + System.currentTimeMillis());
    }
}
```

在这段代码中, `completed()` 方法用于监听并处理 REST 调用成功事件, 见关注点 1。
`failed()` 方法用于监听并处理 REST 调用失败事件, 见关注点 2。

3. 集成测试

最后, 我们启动示例服务, 然后在终端/控制台录入如下 `cURL` 命令来测试 REST 异步处理。示例中的这个资源方法模拟实现了一个耗时的全量图书查询过程, 预期的返回值为全部图书资源的详情。

在示例项目根目录执行 `run.sh`, 启动 REST 服务, 示例如下。

```
pwd
/Users/erichan/sourcecode/jax-rs2-guide-II/samples/5.synchronized/asynchronized
./run.sh
```

使用 `cURL` 命令请求服务, 示例如下。

```
curl :8080/books
```

```

并设计 { HTTP 状态码为 503，服务器端服务不可用 (Response Status 503 SERVICE UNAVAILABLE)
}

"book": [
    {
        "bookId": 10000,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    },
    {
        "bookId": 10001,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    },
    {
        "bookId": 10002,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    },
    {
        "bookId": 10003,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    },
    {
        "bookId": 10004,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    },
    {
        "bookId": 10005,
        "bookName": "Java RESTful Web Services",
        "publisher": "华章 "
    }
]
}

```

4.3 基于 HTTP1.1 的异步通信

本章的前面两节从异步机制，讲述了启用异步的意义和在 REST 中对异步的实现。本节我们从服务器和客户端的通信角度，全面介绍基于 HTTP1.1 协议的异步通信方案。

4.3.1 Polling 技术

服务器—浏览器通信技术的第一种解决方案是客户端轮询技术，即 Polling。

1. 简述

客户端轮询技术 (Polling) 相对其他方案，是最原始、易行的。即浏览器周期性地主动访问服务器的特定地址，以获取服务器端数据状态的变化。通常，在浏览器端使用 JavaScript 脚本启动一个定时任务，该任务向服务器发送请求并获取资源状态。如果服务器

端特定数据发生变化，会将变化信息响应给客户端，客户端使用响应的数据渲染界面，为用户做出及时的反馈。

Polling 异步通信方案可以结合 HATEOAS 和 Web Link 技术，将服务器端的轮询状态地址返回给客户端，如图 4-1 所示，服务器会在接收请求后立即（以 HATEOAS 或者 Web Link 技术）返回给客户端一个查询处理结果的资源地址，并结束这一次的请求—响应流程，HTTP 连接关闭，HTTP 状态码为 202（注意：不是 HTTP 状态 200 OK，202 代表服务器已接受请求但尚未处理）。客户端通过轮询机制，向新的 REST 地址发起请求并获得该处理的进度状态（完成状态为 HTTP 状态码 100，如果请求过期或者资源地址错误则 HTTP 状态码为 404 即找不到），并最终在获取处理完信息后结束轮询。

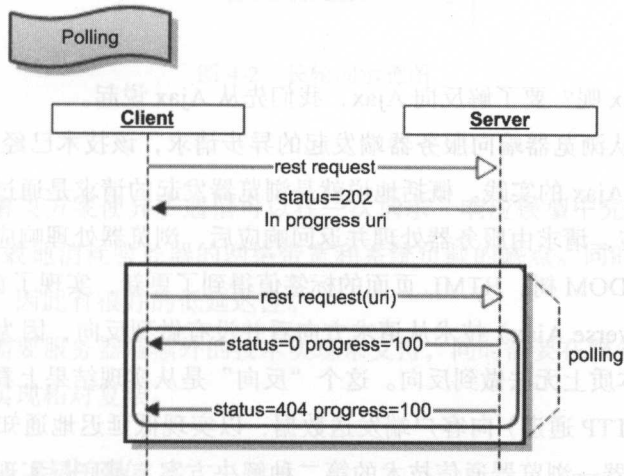


图 4-1 客户端轮询实现异步通信示意图

2. 优点与缺点

优点：这种解决方案比起同步处理的优点是客户端可以即时得到服务器的反馈，并在获得最终结果之前，有机会处理后续业务。另外，Polling 技术不需要对服务器和客户端使用额外的第三方支持包，开发者容易使用现有技术和工具就可以实现。客户端轮询技术对设计没有注入性污染。选择技术架构和设计、实现业务逻辑时这种方式可以即插即拔，不会污染业务平台中结构性的代码。

缺点：客户端轮询技术的缺点是显而易见的，轮询中的每次请求—响应的过程都需要建立新的 HTTP 连接并在结束时关闭该连接。这就造成两大问题。

第一，如果服务器端的业务数据在两次定时任务发起的请求过程中没有变化，后一次请求的做功实际为负数，也就浪费了服务器端的带宽，没有获得有效负载。

第二，也是最让开发者纠结的痛点，即浏览器端的定时器间隔时间参数的设置。由于需要及时获取服务器端的业务数据的状态，这个定时间隔参数设置不宜过长，但是过短又会经常发生第一个问题。因此，间隔时间的设置是个尴尬的坑，因为在编码和调试阶段定义并运行完好的参数，很难和生产环境吻合，甚至开发阶段有可能疏漏或无法覆盖到全部生产环境中的业务场景。还有一个让人难受的地方是这种请求的代码很难抽象出来，因为不同业务的定时间隔都是一个独立的经验值。

4.3.2 Comet 技术

Comet 是反向 Ajax 的技术集，包括长轮询（Long Polling）和流（Streaming）两种技术实现。

1. 简述

什么是反向 Ajax 呢？要了解反向 Ajax，我们先从 Ajax 说起。

Ajax 技术是指从浏览器端向服务器端发起的异步请求，该技术已经烂大街了，5.1.1 节所述的 Polling 就是 Ajax 的实践。概括地说就是浏览器发起的请求是通过脚本实现的，页面并没有提交或者跳转，请求由服务器处理并返回响应后，浏览器处理响应数据并将这一变化渲染到 HTML 中的 DOM 树，HTML 页面的标签值得到了更新，实现了页面的局部刷新。

反向 Ajax（Reverse Ajax）技术从请求方向看并没有做到反向，因为基于请求—响应模式下的 HTTP 请求本质上无法做到反向。这个“反向”是从实现结果上看的，即从服务器端（通过保持连接的 HTTP 通道）向客户端发送数据，以实现低延迟地通知客户端的技术。反向 Ajax 技术是服务器—浏览器通信技术的第二种解决方案，其底层实现依赖于 HTTP 连接不能断开这一前提条件。长轮询（Long Polling）和流（Streaming）技术是反向 Ajax 的两种技术手段，通信原理相同，如图 4-2 所示。

在图 4-2 中，长连接通过 keepAlive 使 HTTP 连接得以保持。为什么要保持连接呢？因为在请求发出后的一定时间内，服务器一直没有做出响应，该连接会因连接超时而断开。Comet 利用 HTTP1.1 的 keepAlive 持久性连接技术，在浏览器发出请求后，通过 keepAlive 保持服务器向浏览器做出响应的通信。这样一来，就解决了连接超时断开的问题。那么，连接的关闭就只有两种情况，一种是浏览器主动断开，一种是服务器端特定数据发生变化，并将这一信息响应给浏览器，主动断开连接完成请求—响应模式的一次请求。

实现 Comet 比起 Polling 要困难得多，服务器端和浏览器端都需要第三方的库来支持这一技术。Atmosphere 库和 CometD 库是实现 Comet 技术的第三方工具包，Jersey 自身并没有提供支持 Comet 实现的包，而是将其交由 Servlet 容器来支持。本书不再对 Comet 的实现做进一步讨论。但读者要清楚 Comet 技术是这个领域的一个选择。

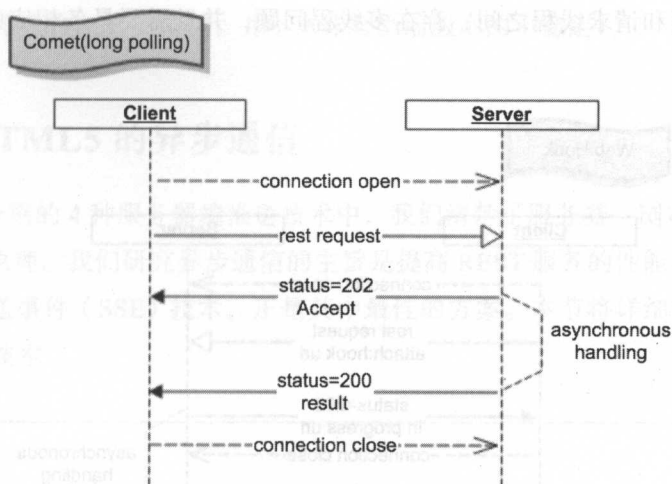


图 4-2 长轮询示意图

2. 优点与缺点

优点：Comet 解决方案使异步通信可以在一次请求—响应模型中完成。反向 Ajax 的技术解决了 Polling 低效地消耗服务器的网络带宽和系统负载的缺点。同时，由于服务器主动向浏览器发送数据，因此有很好的低延迟性。

缺点：Comet 需要服务器端额外的技术实现来支持，同时需要在服务器和浏览器两端引入第三方工具包。实现相对复杂。

4.3.3 Web Hook 异步通信

1. 简述

Web Hook 解决方案是指在客户端发送请求时，将一个回调地址同时发送给服务器，服务器接收响应后，异步处理请求并对本次请求即刻做出响应，客户端随即处理其他业务并监听回调。服务器端在响应客户端后，继续以异步的方式处理方才的请求，在处理完毕后通过回调地址通知客户端处理结果，如图 4-3 所示。

2. 优点与缺点

优点：Web Hook 解决方案具备 Polling 方案的优点，易于实现，无需引入第三方技术，并且没有 Polling 方案中无效的轮询负载。

缺点：Web Hook 这种方案无法在浏览器作为客户端的场景中实施，因为浏览器无从提供一个回调地址给服务器。因此，该方案适用于另外一个服务器做客户端的场景。另外，与随后介绍的 SSE 相比，这种解决方案还是多出了一次服务器回调客户端的 HTTP 连接，并

且客户端回调线程和请求线程之间，存在多线程问题，并且需要具备相应的状态监控机制，需要开发者留意。

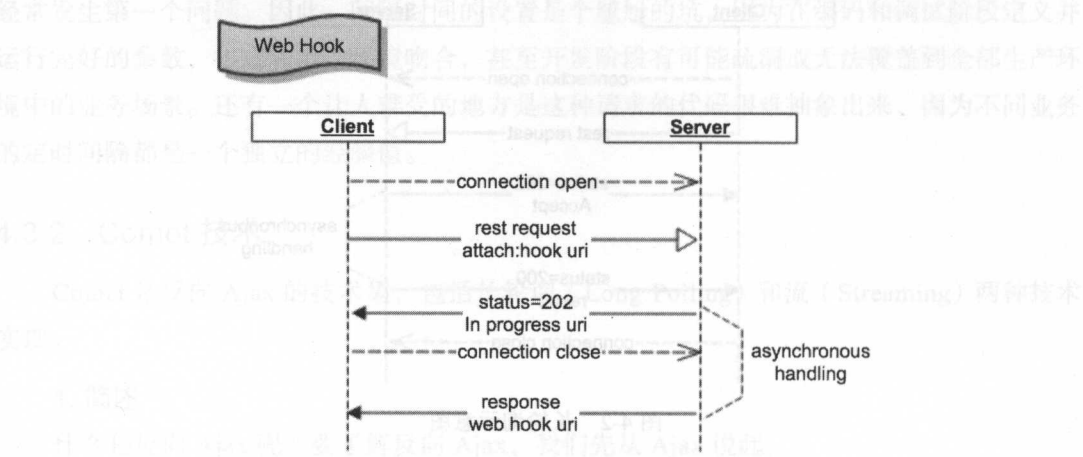


图 4-3 Web Hook 实现异步通信示意图

4.3.4 SSE 技术

SSE (Server-Sent Events) 是 HTML5 技术集的一部分，定义了服务器推送技术的标准规范。

1. 简述

SSE 规范的地址是 <http://dev.w3.org/html5/eventsource>。其核心是基于 EventSource 接口的事件监听机制，包括 onopen、onmessage 和 onerror 3 个事件监听器。SSE 服务器端响应数据的媒体类型 (Content-Type) 是 text/event-stream。Jersey 的媒体库提供了对 SSE 的支持，详述参见 4.4 节。

2. 优点与缺点

优点：SSE 是标准规范——HTML5 标准之一，具备编程语言的无关性。首先 SSE 支持跨语言开发，无论具体使用什么语言和框架，只要按照以 EventSource 接口为中心，完成事件监听机制即可实现 SSE。其次 SSE 支持跨语言的调用，这点很好理解，基于标准接口和标准事件监听，第七层协议上的 HTTP 包很容易被各系统彼此阅读。SSE 的代码实现和交互逻辑相对简单，在 Java EE 生态环境中，得到了 Jersey 提供的支持。

缺点：由于请求—响应模型的限制，SSE 和 Comet 一样，是一种从服务器端到浏览器端的单向通信，浏览器无法在同一条连接上做出二次请求或者对服务器的响应做出“响应”，这一缺点无法支持复杂的交互需求。另外，SSE 标准和 Jersey-sse 支持包都还在不断完善中

(Jersey 提供持续升级的 SSE 支持包，并不意味着当前版本的不稳定)。

4.4 基于 HTML5 的异步通信

从 4.3 节中介绍的 4 种服务器端推送技术中，我们清楚了服务器—浏览器异步通信是什么以及基本实现原理。我们研究异步通信的主旨是提高 REST 服务的性能，而 HTML5 技术栈的服务器端推送事件 (SSE) 技术，正是其中最佳的方案。本节将详细讲述如何在 REST 服务中实现 SSE 技术。

阅读指南

4.4 节的示例源代码地址：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/4.4.sse>。

4.4.1 SSE 的原理

Jersey 的 SSE 支持包 `jersey-media-sse`，基于 HTML5 的 SSE 规范，提供了一套支持 SSE 规范的完整的 API。Maven 坐标定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-sse</artifactId>
</dependency>
```

Jersey 的 SSE 支持包提供两种通信模式，发布—订阅模式和广播模式。前者是一种端到端的通信，后一种是多播通信。接下来我们分别讲述这两种模式。

1. 发布—订阅模式

发布—订阅模式的 API 和典型的工作流程，如图 4-4 所示。

根据 SSE 标准规范定义的 `EventSource` 接口，Jersey SSE 定义了 `EventListener` 接口及其实现类 `EventSource`。SSE 的实现流程描述如下。

第一步，在客户端创建 `EventSource` 实例并覆盖 `onEvent()` 方法。该方法用于处理服务器端推送的事件，输入参数为 `InboundEvent` (进站事件)。`EventSource` 的构造函数包含一个输入参数，是 `WebTarget` 端点，指向服务器端的资源路径为 `sse` 的 GET 方法，服务器端的这个资源方法会使用注解 `@Produces(SseFeature.SERVER_SENT_EVENTS)`。如图 4-4 中 1 所示，在 `EventSource` 实例化过程中，客户端会从这个端点向服务器发出请求，并在请求头中指定接收数据的媒体类型为 `SseFeature.SERVER_SENT_EVENTS`，即 `Accept` 头信息声明为 `text/event-stream` 类型。

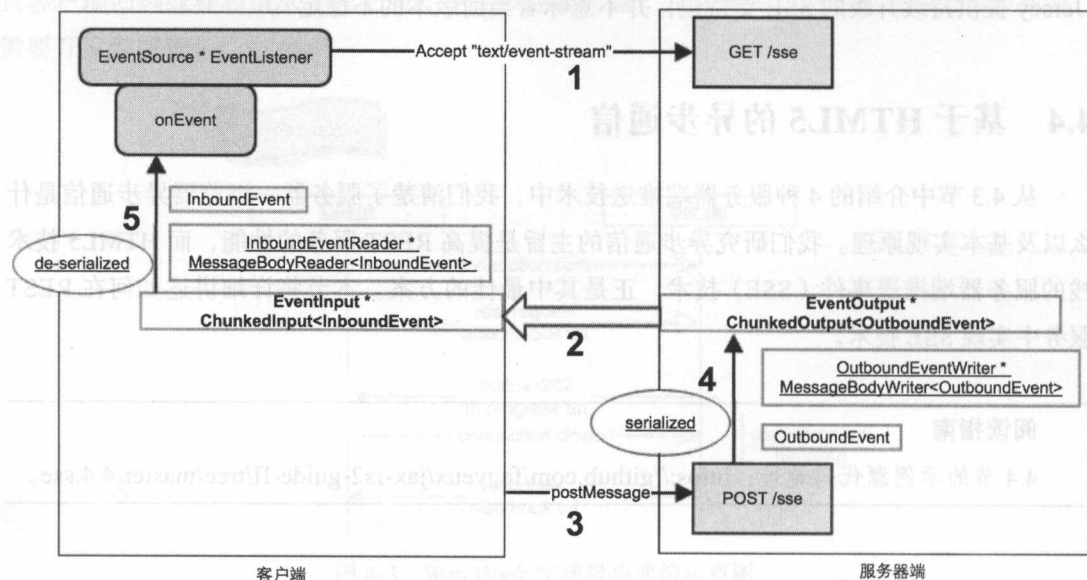


图 4-4 Jersey 实现 SSE 流程示意图

第二步，请求被服务器接收后，开启 SSE 事件通信通道，方向是从服务器端向客户端并返回响应给客户端。此时，HTTP 连接被保持着，并没有关闭，如图 4-4 中 2 所示。

此时，在 SSE 事件通信通道的客户端一端，会建立 EventInput 信道，用于读取 InboundEvent（进站事件）。EventInput 类继承自 ChunkedInput，ChunkedInput 允许将数据在一条信道中分次传输；EventInput 类的泛型类型为 InboundEvent，即上述的 onEvent 方法待处理的类型。当进站事件到达时，MessageBodyReader 的 SSE 实现类 InboundEventReader 会解析数据，并反序列化为 InboundEvent 类型的数据。

同时，在 SSE 事件通信通道的服务器一端，会建立 EventOutput 信道，用于写入 OutboundEvent（出站事件）。EventOutput 类继承自 ChunkedOutput，ChunkedOutput 允许在发送出站事件后，HTTP 连接通过 HTTP1.1 的 Keep-Alive 保持连接，出站事件被写入 HTTP 响应头后，EventOutput 信道将等待更多的服务器推送事件。出站事件的序列化写入，由 MessageBodyWriter 的 SSE 实现类 OutboundEventWriter 完成。

接下来的三个步骤就是在这个信道上完成的。

第三步，客户端向服务器发送 POST 请求，如图 4-4 中 3 所示。

第四步，服务器端接收后，会向 EventOutput 信道写入数据，如图 4-4 中 4 所示。

最后一步，客户端监听到信道中有数据到达，将读取并处理推送事件，如图 4-4 中 5 所示。

到此，服务器推送事件的流程走完一遍。信道的连接可以由服务器主动关闭或者由客户端请求关闭。

2. 广播模式

通过上述的发布—订阅模式，我们对 Jersey 的 SSE 支持包和通信机制有了几乎全面的了解。广播模式与发布—订阅模式相比，客户端的实现相同，服务器端推送事件的写入，由端到端的 EventInput 信道，换成了多点广播类 SseBroadcaster。SseBroadcaster 类继承自 Broadcaster 类，泛型类型为 OutboundEvent（出站事件）。SseBroadcaster 类提供了一次关闭多点信道的方法 closeAll()，可以根据业务需要，在完成广播事件后执行。

到此，Jersey 的 SSE 支持包中的成员类都已经一一介绍，如图 4-5 所示。接下来将讲述如何使用 Jersey 的 SSE 支持包来实践上述两种模式。

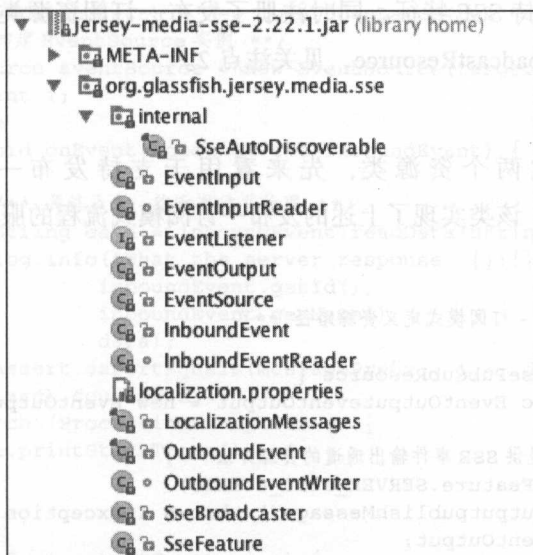


图 4-5 Jersey 的 SSE 实现包示意图

4.4.2 发布—订阅模式的实现

我们的 REST 应用实践，可以概括为 2 个环节。第一个环节是基于本章上述理论开发 REST 入口类 Application、资源类以及单元测试类；第二个环节是集成测试，以验证资源方法的功能。

1. Application 类

首先，我们介绍支持 SSE 的 REST 应用入口类 AirResourceConfig，该类是 Application

类的子类。示例代码如下。

```
/** 关注点 1: 为 SSE 定义资源路径 */
@ApplicationPath("/event/*")
public class AirResourceConfig extends ResourceConfig {
    public AirResourceConfig() {
        /** 关注点 2: 注册 Feture 和资源类 */
        super(
            SseFeature.class,
            AirSsePubSubResource.class,
            AirSseBroadcastResource.class
        );
    }
}
```

在这段代码中，定义了支持 SSE 的 REST 服务根资源路径：/event/*，见关注点 1。在构造方法中，我们手动注册了 SseFeature 类（在 jersey2.8 版本之后可以自动探测），用以标识我们的 REST 服务支持 SSE 特征。同时注册了发布—订阅资源类 AirSsePubSubResource 和广播资源类 AirSseBroadcastResource，见关注点 2。

2. 资源类

我们分别讲述这两个资源类，先来看用于支持发布—订阅模式的资源类 AirSsePubSubResource。该类实现了上述的发布—订阅模式流程的服务器端代码，示例代码如下。

```
/** 关注点 1: 为发布 - 订阅模式定义资源路径 */
@Path("pubsub")
public class AirSsePubSubResource {
    private static EventOutput eventOutput = new EventOutput();
    @GET
    /** 关注点 2: 提供 SSE 事件输出通道的资源方法 */
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput publishMessage() throws IOException {
        return eventOutput;
    }
    @POST
    public void saveMessage(String message) throws IOException {
        /** 关注点 3: 执行业务逻辑 */
        log.info("What the client post: {}", message);
        /** 关注点 4: 写入 SSE 通道的资源方法 */
        eventOutput.write(new OutboundEvent.Builder()
            .id(System.nanoTime() + "")
            .name("post-message")
            .data(String.class, message).build());
    }
}
```

在这段代码中，资源地址定义了发布—订阅模式的资源地址为 "pubsub"，见关注点 1。GET 方法用于公布 SSE 通信通道，POST 方法用于处理业务和写入 SSE 事件，两者逻辑上

是先后被调用的关系。publishMessage 方法公布了推送事件输出通道的接口，返回值是前述的 EventOutput 类型的推送事件输出信道，见关注点 2。POST 方法 saveMessage() 用于接收并处理客户端提交的信息，见关注点 3，最后并响应信息写入推送事件输出信道，根据 HTML5 的 SSE 规范，出站事件 OutboundEvent 的数据结构包含 3 个主要信息：id、name 和 data，见关注点 4。

3. 测试类

服务器端的代码已经完成，我们再通过一个单元测试类 SsePubSubTest 实现客户端的代码。SsePubSubTest 类继承了 Jersey 的测试框架类 JerseyTest（参见第 6 章），省去了样板测试代码，示例代码如下。

```
@Test
public void testEventSource() throws InterruptedException, URISyntaxException {
    final CountDownLatch latch = new CountDownLatch(testCount);
    /** 关注点 1: 构建 EventSource 实例 */
    final EventSource eventSource = new EventSource(target().path(ROOT_PATH)) {
        private int i;
        @Override
        public void onEvent(InboundEvent inboundEvent) {
            try {
                /** 关注点 2: 监听事件并处理 */
                String data = inboundEvent.readData(String.class);
                log.info("What the server response: {}:{}:{}",
                    inboundEvent.getId(),
                    inboundEvent.getName(),
                    data);
                Assert.assertEquals(messagePrefix + i++, data);
                latch.countDown();
            } catch (ProcessingException e) {
                e.printStackTrace();
            }
        }
    };
    for (int i = 0; i < testCount; i++) {
        target().path(ROOT_PATH).request().post(Entity.text(messagePrefix + i));
    }
    try {
        latch.await();
    } finally {
        eventSource.close();
    }
}
```

在这段代码中，集成测试方法 testEventSource() 模拟了客户端监听并处理 SSE 事件，并检验了 SSE 发布—订阅模式的功能和流程。

首先，客户端创建了 EventSource 实例，并通过该实例请求服务器端的 GET 方法，以获得 SSE 事件输出信道，见关注点 1。在 EventSource 接口的实现中，onEvent 方法中做了

3 件事：第一是打印输出服务器推送事件的内容，内容包括前面讲述的 SSE 规范定义的数据结构：id、name 和 data。第二是使用相等断言测试服务器端返回数据是否符合预期。第三是调用同步器 `CountDownLatch` 实例的 `countDown` 方法（原因见下文），见关注点 2。

测试代码中使用 `CountDownLatch` 的原因是在测试流程中，通过 GET 订阅服务器 SSE 事件通知和通过 POST 发送数据是两个异步操作，而我们期待的顺序是在执行完 POST 代码块后，主线程不退出，以便前面代码块中定义的监听和处理推送事件的 `onEvent()` 方法可以被执行。为此，我们引入 `CountDownLatch` 实例并调用了其 `await()` 方法，使得测试主线程被阻塞而不是直接走完退出，以便等待回调被执行。相应地，`onEvent()` 方法的第三步是调用 `CountDownLatch` 实例的 `countDown` 方法来递减其内部计数器，确保我们期待的消息都被接收后，主线程最终被唤醒以结束整个测试流程。

最后，我们来看一下完整的单元测试输出。

```
18:26:24.183 AirSsePubSubResource - What the client post: pubsub-0
18:26:24.196 SsePubSubTest - What the server response: 30693005931717:post-
message:pubsub-0
18:26:24.199 AirSsePubSubResource - What the client post: pubsub-1
18:26:24.201 SsePubSubTest - What the server response: 30693020207612:post-
message:pubsub-1
18:26:24.205 AirSsePubSubResource - What the client post: pubsub-2
18:26:24.207 SsePubSubTest - What the server response: 30693026189883:post-
message:pubsub-2
18:26:24.211 AirSsePubSubResource - What the client post: pubsub-3
18:26:24.213 SsePubSubTest - What the server response: 30693032364560:post-
message:pubsub-3
18:26:24.218 AirSsePubSubResource - What the client post: pubsub-4
18:26:24.219 SsePubSubTest - What the server response: 30693039008185:post-
message:pubsub-4
18:26:24.223 AirSsePubSubResource - What the client post: pubsub-5
18:26:24.224 SsePubSubTest - What the server response: 30693043716912:post-
message:pubsub-5
18:26:24.228 AirSsePubSubResource - What the client post: pubsub-6
18:26:24.229 SsePubSubTest - What the server response: 30693048605712:post-
message:pubsub-6
18:26:24.233 AirSsePubSubResource - What the client post: pubsub-7
18:26:24.235 SsePubSubTest - What the server response: 30693054086320:post-
message:pubsub-7
18:26:24.240 AirSsePubSubResource - What the client post: pubsub-8
18:26:24.241 SsePubSubTest - What the server response: 30693060768748:post-
message:pubsub-8
18:26:24.246 AirSsePubSubResource - What the client post: pubsub-9
18:26:24.247 SsePubSubTest - What the server response: 30693066859047:post-
message:pubsub-9
```

4. 集成测试

集成测试更为普遍的方式是使用 `cURL` 命令直接对资源地址进行访问，即时检测资源方法的功能，或者使用 `shell` 脚本编写 `cURL` 请求。本例将使用 `cURL` 命令分别对上述的两

种模式的实现进行集成测试。

启动示例服务，然后使用两个终端/控制台测试发布—订阅流程。在第一个终端中录入如下 cURL 命令，然后等待其输出。这一步请求了 GET 方法并建立了 HTTP 连接。

```
curl -H "Accept: text/event-stream" --url http://localhost:8080/sse/event/pubsub
在第二个终端中依次录入如下 cURL 命令，以模拟多次提交 POST 请求的流程。
curl -X POST --data "Java1.5" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.6" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.7" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.8" --url http://localhost:8080/sse/event/pubsub
```

第一个终端应输出如下信息。展示了服务器端在处理每一次 POST 请求时，向事件输出通道写入出站事件。

```
event: post message
id: 22127543063654
data: Java1.5
event: post message
id: 22134553708520
data: Java1.6
event: post message
id: 22141048454287
data: Java1.7
event: post message
id: 22149255904866
data: Java1.8
```

4.4.3 广播模式的实现

有了发布—订阅模式的实践，广播模式就很好理解了。广播模式依然按资源类、测试类和集成测试顺序讲述，并省去与发布—订阅相同的信息。

1. 资源类

资源类 `AirSseBroadcastResource` 用于支持广播模式的 SSE，示例代码如下。

```
/** 关注点 1: 为广播模式的 SSE 定义资源路径 */
@Path("broadcast")
public class AirSseBroadcastResource {
    private static final BlockingQueue<BroadcastProcess> processQueue =
        new LinkedBlockingQueue<>(1);
    @Path("book")
    @POST
    public Boolean postBook(@DefaultValue("0") @QueryParam("total") int total,
        String bookName) {
        /** 关注点 2: 调用 BroadcastProcess 实例实现广播 */
        final BroadcastProcess broadcastProcess = new BroadcastProcess(total, bookName);
        processQueue.add(broadcastProcess);
        Executors.newSingleThreadExecutor().execute(broadcastProcess);
        return true;
    }
}
```

/** 关注点 3: 广播处理线程类 */

```
static class BroadcastProcess implements Runnable {
    .....
    public void run() {
        .....
        OutboundEvent.Builder eventBuilder = new OutboundEvent.Builder()
            .mediaType(MediaType.TEXT_PLAIN_TYPE);
        OutboundEvent event = eventBuilder.id(processId + "")
            .name("New Book Name")
            .data(String.class, bookName).build();
        broadcaster.broadcast(event);
    }
}
```

在这段代码中, 资源地址定义了广播模式的资源地址为 "broadcast", 见关注点 1。实现广播的核心代码块位于内部线程类 BroadcastProcess 中, 该线程类使用广播类 SseBroadcaster 实例将 SSE 事件广播出去, 见关注点 3。

POST 方法 postBook() 资源地址是 broadcast/book 用于接收客户端提交的收听广播的客户端数量和最新图书信息。一旦接收数据, 该方法会动态生成一个线程类 BroadcastProcess 的实例, 并尝试将其缓存到 BlockingQueue 实例中。BlockingQueue 的 add() 方法会在尝试入队失败时, 直接抛出异常, 意味着广播类只缓存最新的一条图书资源, 直接失败的处理可以避免客户端请求线程阻塞/挂起, 见关注点 2。缓存队列会被 GET 方法消费, 篇幅所限没有展示全部代码, 读者可以从源代码中查阅。

2. 测试类

广播测试类 SseBroadcaseTest 的示例代码如下。与前一测试非常类似不冗述相同部分, 需要额外说明的是, 在每一个客户端的 GET 请求中, 生成 EventSource 类实例的过程是通过 EventSource 类的内部 Builder 完成的, 因此并没有自动打开 SSE 信道, 因此在注册监听、覆盖 onEvent() 方法后, 需要显式调用 open() 方法, 见关注点 1。

```
readerEventSources[i] = EventSource.target(endpoint).build();
readerEventSources[i].register(new EventListener() {
    @Override
    public void onEvent(InboundEvent inboundEvent) {
        try {
            String data = inboundEvent.readData(String.class);
            log.info("What the server response: {}:{}:{}",
                inboundEvent.getId(),
                inboundEvent.getName(),
                data);
            Assert.assertEquals(newBookName, data);
            doneLatch.countDown();
        } catch (ProcessingException e) {
            log.error("", e);
        }
    }
});
```

```
/** 关注点 1: 显式调用 open() 方法 */
readerEventSources[i].open();
```

3. 集成测试

启动服务，然后根据上述测试场景的逻辑，按照图 4-6 所示，使用 3 个终端 / 控制台测试广播流程。

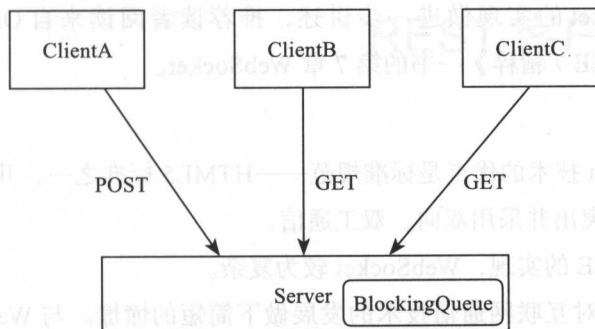


图 4-6 SSE 广播测试示意图

在图 4-6 所示的第一终端 ClientA 中录入如下 cURL 命令，其中，数据为 jax-rs2-guide；参数 total=2，当两个客户端连接到服务器后，向全部客户端发起广播。

```
curl -X POST --data "jax-rs2-guide" http://localhost:8080/sse/event/broadcast/
book?total=2
```

接下来，在第二个终端 ClientB 录入如下 cURL 命令，终端会出现阻塞，但不去管它，继续第三个终端 ClientC 的命令录入。

```
curl -H "Accept: text/event-stream" http://localhost:8080/sse/event/broadcast/
book?clientId=1
```

在第三个录入如下 cURL 命令，第二、第三个会同时收到服务器的广播消息。

```
curl -H "Accept: text/event-stream" http://localhost:8080/sse/event/broadcast/
book?clientId=2
```

事件消息中的 data 信息既是 POST 提交的内容。

```
event: New Book Name
id: 33035275182692
data: jax-rs2-guide
```

4.4.4 WebSocket 技术

HTML5 包含了 SSE 和 Web Socket。SSE 用于服务器推送事件，但并不仅限于这样使用，本节前述已经展示了其处理异步通信的能力。Web Socket 无疑非常适用于服务器推送

的场景。WebSocket (RFC 6455) 是 HTML5 技术集的一部分, 它提供了一个双向的、在一条 TCP 信道中的客户端和服务器之间全双工的通信。

1. 简述

WebSocket 消除了所有与 HTTP 连接的无状态特性相关的限制。Java EE 7 已经支持 WebSocket, 其参考实现是 Glashfish 项目的 Tyrus 子项目 (项目地址: <https://tyrus.java.net>)。本书不再对 WebSocket 的实现做进一步讲述, 推荐读者阅读来自 Oracle 的布道者 Arun Gupta 所著的《Java EE 7 精粹》一书的第 7 章 WebSocket。

2. 优点与缺点

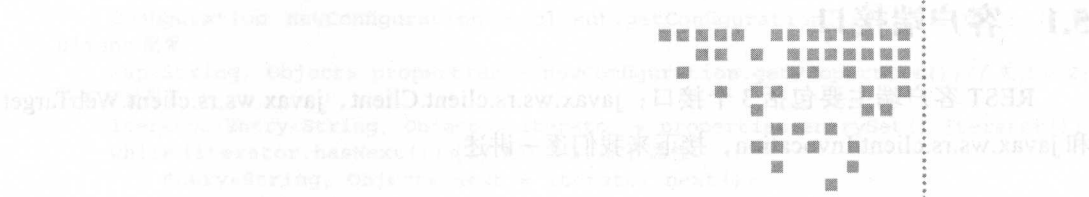
优点: WebSocket 技术的优点是标准规范——HTML5 标准之一, 并逐渐成为流行趋势。它的功能强大、性能突出并采用双向、双工通信。

缺点: 相对于 SSE 的实现, WebSocket 较为复杂。

最后, 笔者在此对互联网通信技术的发展做下简短的憧憬。与 WebSocket 非常类似的标准是 HTTP2 (前身是 SPDY), 两者的共同点是解决 HTTP1.1 无法双向通信的问题。所不同的是, 前者借助 HTTP1.1 握手, 而后者是对 HTTP1.1 的全面升级。技术日新月异, 我们当下讨论的异步通信、服务器推送技术都是基于 HTTP1.1 协议的, REST 服务技术必将会随着新技术的发展而产生新的解决方案。

4.5 本章小节

本章讲述了 REST 服务的异步机制和实践。首先分析了服务器端和客户端采用异步方式可以解决哪些问题, 接下来分别给出两端基于 JAX-RS2 的实践。随后, 讲述了基于 HTTP1.1 的实现技术以及 HTML5 的异步通信实践。



REST 客户端

在 Java API for RESTful Web services (JAX-RS) 1.x 的定义中，没有关于 REST 客户端的定义。但是，为了方便开发者使用，JAX-RS 的多个实现项目（包括 Jersey、CXF 等）都自行弥补了 REST 客户端的实现。Jersey1.x 的客户端实现底层采用的是 Apache Http Client。JAX-RS2 标准化了 Client API（客户端 API）。Jersey2.x 的客户端包 jersey-client 实现了 JAX-RS2 的客户端 API，并对其进行了可插拔的（pluggable）扩展。客户端 API 通过 HTTP 协议请求 Web 资源，其设计宗旨是使客户端 API 符合统一接口和 REST 架构风格，同时客户端 API 应该易于使用，而且服务器端在概念和扩展点上保持一致。与 Apache HTTP Client 和 HttpURLConnection 相比，客户端 API 是可感知 REST 的高层 API，可以与 Providers 集成，返回值直接对应高层的业务类实例，而不是 JAXB 对象或者更为低层的数据类型。

有关 REST 客户端的实例贯穿每一章的实例中，因为本书提倡单元测试，每个实例中的单元测试皆使用了 REST 客户端。比如有关安全性的客户端实例可以参考相关章节的代码，本章实例主要针对通用接口和连接器实践。

阅读指南

本章示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/5.jaxrs2-client>。

5.1 客户端接口

REST 客户端主要包括 3 个接口：javax.ws.rs.client.Client、javax.ws.rs.client.WebTarget 和 javax.ws.rs.client.Invocation，接下来我们逐一讲述。

5.1.1 Client 接口

Client 接口是 REST 客户端的基本接口，用于和 REST 服务器通信。Client 被定义为一种重量级的对象，其内部要管理客户端通信底层实现所需的各种对象，比如连接器、解析器等。因此，不推荐在应用中产生大量的 Client 实例，这一点需要开发者小心，切记。为示警戒，摘出源码中的注释如下。

```
It is therefore advised to construct only a small number of Client instances
in the application.
```

这句话的意思是在应用中尽量少地构造 Client 实例。另外，该接口要求其实例要有关闭连接的保障，否则会造成内存泄漏。

Jersey 对 JAX-RS2 的 Client 接口的实现类是 org.glassfish.jersey.client.JerseyClient。创建一个 Client 实例是通过 ClientBuilder 构造的，通常使用一个 ClientConfig 实例作为参数，该实例的构建过程是 23 种设计模式中构造模式的实践。在传入 ClientConfig 实例前，通常会注册（register）相关的 Provider 和 Feature，也可以设定自定义属性（键值对），但这些编码都是可选的。最简单的形式是没有参数的构造，示例如下。

```
Client client = ClientBuilder.newClient();
```

通常情况下，客户端需要加载指定的资源、注册某些 provider 或者预定义某些属性，以支持更复杂的业务逻辑。我们在 ClientConfig 实例中完成了这些配置后，将其作为参数并在 Client 构建时传入，示例如下。

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.register(MyProvider.class); // 关注点 1: 注册 provider
clientConfig.register(MyFeature.class); // 关注点 2: 注册 Feature
clientConfig.register(new AnotherClientFilter()); // 关注点 3: 注册 Filter
Client client = ClientBuilder.newClient(clientConfig);
client.property("MyProperty", "MyValue"); // 关注点 4: 注册属性
```

在这段代码中，在构造 Client 实例前，预先配置了 ClientConfig 实例，见关注点 1 ~ 3。在 Client 实例中，通过 property 方法同样可以配置相关的属性，见关注点 4。

在配置完毕后，可以通过 Client 的 getConfiguration() 方法获取当前 Client 实例的配置信息，示例如下。

```
private void checkConfig() {
```

```

Configuration newConfiguration = client.getConfiguration();//关注点1: 获取
client 配置
Map<String, Object> properties = newConfiguration.getProperties();//关注点2:
获取配置属性
Iterator<Entry<String, Object>> iterator = properties.entrySet().iterator();
while (iterator.hasNext()) { // 关注点3: 迭代属性
    Entry<String, Object> next = iterator.next();
    LOGGER.debug (next.getKey() + ":" + next.getValue());
}
}

```

在这段代码中，在关注点1处获取了 client 配置。该配置包含了当前客户端实例的属性键值对信息，见关注点2。通过迭代该键值对可以获取全部的属性信息，见关注点3。

Client 接口还提供了对客户端的安全连接和异步的支持。为了不重复讲述，请详见本书的第6章和第8章的相关内容。

5.1.2 WebTarget 接口

WebTarget 接口是为 REST 客户端实现资源定位的接口。通过 WebTarget 接口，我们可以定义请求资源的具体地址、查询参数和媒体类型等信息。Jersey 对 JAX-RS2 的 WebTarget 接口的实现类是 org.glassfish.jersey.client.JerseyWebTarget。

WebTarget 对象接收配置参数的方法定义有很浓厚的 DSL (Domain Specific Language) 味道，通过方法链即可完成对一个 WebTarget 实例的配置。但是，值得注意的是，如果不将方法链写成一行，而是要分开来写，需要知道 WebTarget 接口所采用的方法链模式是一个不变式 (immutable)。也就是说，其返回值是一个新的 WebTarget 对象，我们无法像使用 StringBuilder 的 append 方法 (该方法返回的是 this) 那样设置 WebTarget 对象，而是必须接收设置方法的返回值，作为后续流程的句柄。

如下做法是想当然的，WebTarget 初始化一行后面的方法调用并不对该 WebTarget 实例产生作用。

```

WebTarget webTarget = client.target (BASE_URI);
webTarget.path ("books");//关注点1: 设置的返回值被丢弃了
webTarget.path ("book");
webTarget.queryParam ("bookId", "1");
final Invocation.Builder invocationBuilder = webTarget.request (MediaType.
APPLICATION_XML);

```

在这段代码中，WebTarget 实例每次设置都将产生一个新的 WebTarget 实例，但不幸的是该实例在下一行没有被使用，原来的 WebTarget 实例并没有使用到这一配置，见关注点1。正确的用法如下，每次使用一个新的 WebTarget 都接收 WebTarget 的方法调用，最终使用最后一个实例作为请求句柄。

```
WebTarget webTarget = client.target(BASE_URI);
WebTarget pathTarget = webTarget.path("books");
WebTarget pathTarget2 = pathTarget.path("book");
WebTarget queryTarget = pathTarget2.queryParam("bookId", "1");
final Invocation.Builder invocationBuilder =
    queryTarget.request(MediaType.APPLICATION_XML);
```

5.1.3 Invocation 接口

Invocation 接口是在完成资源定位配置后，向 REST 服务端发起请求的接口。请求包括同步和异步两种方式，由 Invocation 接口内部的 Builder 接口定义，Builder 接口继承了同步接口 SyncInvoker。SyncInvoker 中定义了标准的 HTTP 请求方法。Jersey 对 JAX-RS2 的 Invocation 接口的实现类是 `org.glassfish.jersey.client.JerseyInvocation`。

Invocation.Builder 接口实例分别执行了 GET 请求和 POST 请求来提交查询和创建。默认情况下，HTTP 方法调用的返回类型是 Response 类型，同时也支持泛型类型的返回值。

```
final Invocation.Builder invocationBuilder =
    queryTarget.request(MediaType.APPLICATION_XML);
final Book book = invocationBuilder.get(Book.class);
Invocation.Builder invocationBuilder = target().path(USER_PATH).request();
Response response = invocationBuilder.post(userEntity);
```

通常，我们希望返回的类型为 POJO，其内包含基本类型、List 或者 Map。比如 Books 类内部的 Book 集合字段 `List<Book>` 就是这样的一种封装，我们可以很方便地通过 Invocation.Builder 的 `get` 方法获取 POJO 实例。

但是，如果某些异构的服务端只能返回 List 类型的 POJO 集合数据，我们将无法简便地使用 Books 这样的封装类型。此时，我们可以使用 `GenericType` 来解决，示例代码如下。

```
Invocation.Builder invocation = webTarget.request();
GenericType<List<Book>> genericType = new GenericType<List<Book>>() {
};
List<Book> books = invocation.get(genericType);
for (Book book : books) {
    System.out.println(book);
}
```

5.2 连接池

5.2.1 资源释放

作为 REST 框架，JAX-RS2 不希望开发者编码实现对客户端实例的资源管理，Response 实例的 `readEntity()` 在返回响应实体的同时，即完成了对客户端资源的释放。因此，开发者

无须担心连接释放等资源管理细节。也许开发者的编码没有使用 `Response` 的 `readEntity()`，而是直接返回泛型类型的对象，但是其底层还是使用了这个方法。我们可以从图 5-1 的调用栈示意图中一窥究竟。

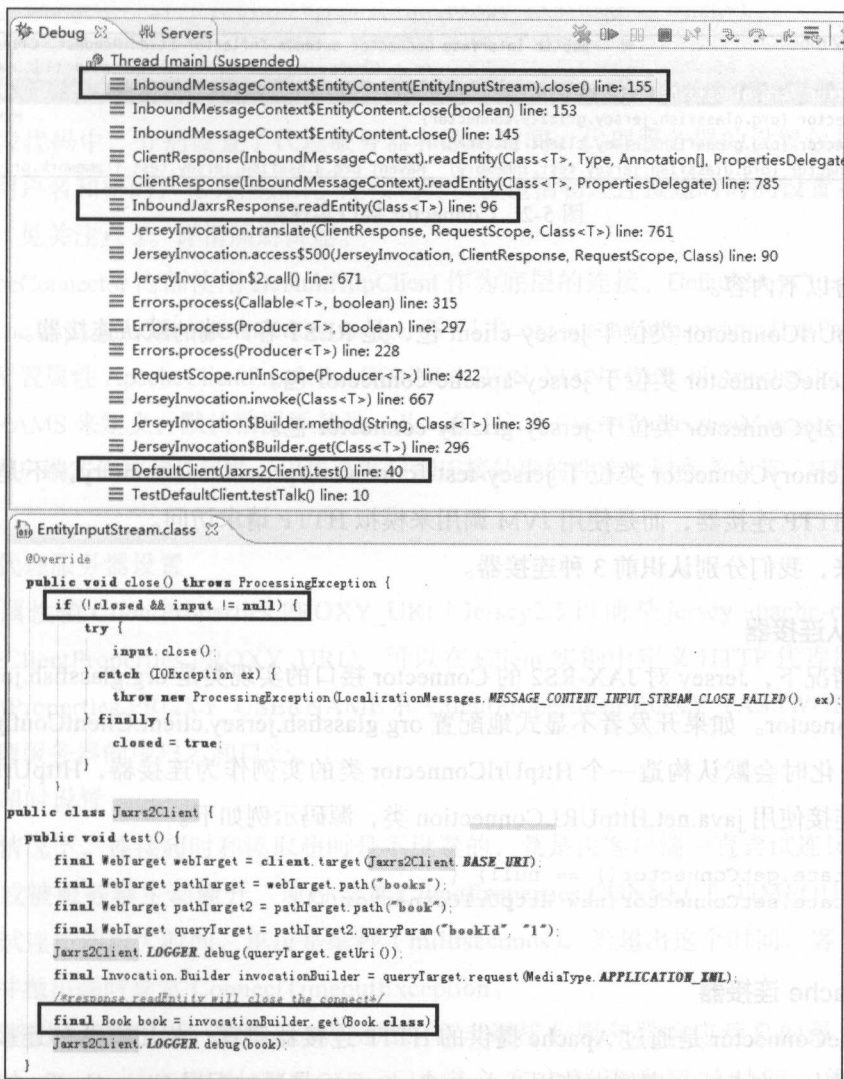


图 5-1 REST 请求连接资源释放

在图 5-1 中，测试代码使用 `Invocation.Builder` 接口实例执行 GET 请求，根据 DEBUG 栈信息，可以发现 `readEntity()` 方法的身影，对应显示 96 行的一行。再往下的栈进入了输入流类中，对应显示 155 行的一行。在该类的 `close()` 方法中，可以看到最终关闭了流对象，资源得以释放。

5.2.2 连接器

Connector 接口是 REST 客户端底层连接器接口，Jersey 为 Connector 接口提供了 4 个实现，如图 5-2 所示。

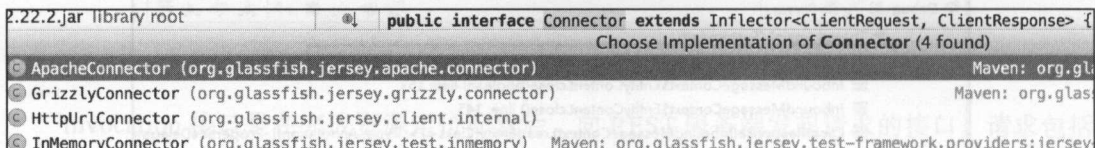


图 5-2 Connector 接口实现类

其中含以下内容。

- HttpURLConnection 类位于 jersey-client 包，是 REST 客户端的默认连接器。
- ApacheConnector 类位于 jersey-apache-connector 包。
- GrizzlyConnector 类位于 jersey-grizzly-connector 包。
- InMemoryConnector 类位于 jersey-test-framework-provider-inmemory，不是一种真实的 HTTP 连接器，而是使用 JVM 调用来模拟 HTTP 请求访问。

接下来，我们分别认识前 3 种连接器。

1. 默认连接器

默认情况下，Jersey 对 JAX-RS2 的 Connector 接口的实现类是 org.glassfish.jersey.client.HttpUrlConnector。如果开发者不显式地配置 org.glassfish.jersey.client.ClientConfig，那么在 Client 初始化时会默认构造一个 HttpURLConnection 类的实例作为连接器，HttpUrlConnector 类底层的连接使用 java.net.HttpURLConnection 类，源码示例如下。

```

if (state.getConnector() == null) {
    state.setConnector(new HttpURLConnection());
}
  
```

2. Apache 连接器

ApacheConnector 是通过 Apache 提供的 HTTP 连接器实现，相比默认的连接器的功能更完整、更强大。在 Client 实例中使用 ApacheConnector 是通过配置 ClientConfig 的 connector 来实现的，示例如下。

```

final ApacheConnector connector = new ApacheConnector(clientConfig);
clientConfig.connector(connector);
client = ClientBuilder.newClient(clientConfig);
  
```

在这段代码中，ClientConfig 实例在 Client 加载前，完成了对 connector 的配置，该 connector 是一个 ApacheConnector 实例。ClientConfig 实例还可以配置和 Apache 连接器相

关的属性，示例如下。

```
// 关注点 1: 代理服务器配置
clientConfig.property(ClientProperties.PROXY_URI, "http://192.168.0.100");
clientConfig.property(ClientProperties.PROXY_USERNAME, "erichan");
clientConfig.property(ClientProperties.PROXY_PASSWORD, "han");
// 关注点 2: 连接超时设置
clientConfig.property(ClientProperties.CONNECT_TIMEOUT, 1000);
clientConfig.property(ClientProperties.READ_TIMEOUT, 2000);
```

在这段代码中，分别设置了代理服务器和超时时间。代理服务器的设置包括代理服务器地址、用户名和密码，见关注点 1。连接超时设置包括物理连接超时时间设置和读取数据超时设置，见关注点 2。详情随后讲述。

ApacheConnector 内部使用 DefaultHttpClient 作为底层的连接。DefaultHttpClient 内部定义了 org.apache.http.conn.ClientConnectionManager 接口和 org.apache.http.params.HttpParams 接口，分别通过配置属性 ApacheClientProperties.CONNECTION_MANAGER 和 ApacheClientProperties.HTTP_PARAMS 来定义，默认情况下都是 null。通过定义 ClientConnectionManager 接口的实例可以改变客户端实例加载的策略，以提高客户端连接处理的性能。请参考本节“HTTP 连接池”相关内容。

(1) 代理服务器设置

通过属性值 ClientProperties.PROXY_URI (Jersey2.5 以前是 jersey-apache-connector 包的 ApacheClientProperties.PROXY_URI)，可以在 Client 实例中定义 HTTP 代理服务器的地址。ClientProperties.PROXY_USERNAME 和 ClientProperties.PROXY_PASSWORD 分别用于配置代理服务器的用户名和口令。

(2) 超时设置

默认情况下，连接超时和读取超时是不设置的，就是说客户端一直尝试连接或者读取，直到成功或被服务器主动断开。通过设置 ClientProperties.CONNECT_TIMEOUT 可以定义客户端尝试连接的最大时间，单位是毫秒 (milliseconds)，当超出这个时间，客户端会主动放弃连接并抛出超时异常 ConnectTimeoutException。

读取超时是指连接和资源定位成功后，客户端接收服务器响应消息的最大时间。通过设置 ClientProperties.READ_TIMEOUT 可以定义客户端读取超时时间，单位是毫秒 (milliseconds)，当超出这个时间，客户端会主动关闭连接，抛出 SocketTimeoutException 异常。

3. Grizzly 连接器

GrizzlyConnector 是 Grizzly 提供的连接器实现，其内部使用异步处理客户端 com.ning.http.client.AsyncHttpClient 类作为底层的连接，该类来自于 AsyncHttpClient 项目 (地址是 <https://github.com/AsyncHttpClient>)。该项目的 2.0 版本将 Maven 的坐标信息进行了更改：

groupId 从 com.ning.http.client 改为 org.asynchttpclient, com.ning.http.client 包名改为 org.asynchttpclient。读者在引用 Grizzly 连接器时应当注意所用版本的 Maven 坐标是否符合上述定义。

AsyncHttpClient 使用 GrizzlyAsyncHttpProvider 处理 HTTP 请求异步事件, GrizzlyAsyncHttpProvider 是 AsyncHandler 接口的实现类。

在 Client 实例中使用 GrizzlyConnector 是通过配置 ClientConfig 的 connector 来实现的, 示例如下。

```
final GrizzlyConnector connector = new GrizzlyConnector(clientConfig);
clientConfig.connector(connector);
client = ClientBuilder.newClient(clientConfig);
```

在这段代码中, ClientConfig 在 Client 加载前, 配置 connector 为 GrizzlyConnector 实例。

5.2.3 HTTP 连接池

在讲述完 4 种连接器后, 我们回到 Client 是重型组件的这个问题。

Client 的内部不只是简单地包含一个 HTTP 连接器的设计, 而是携带了诸如资源状态等信息, 因此在系统中频繁地创建 Client 实例会影响总体的性能。一种常见的解决方案是使用 HTTP 连接池来管理连接, 而不是每次请求都创建一个 Client 实例。

这里使用 ApacheConnector 来实现 HTTP 连接池, 示例如下。

```
final ClientConfig clientConfig = new ClientConfig();
final PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();
// 关注点 1: 配置连接池管理实例
cm.setMaxTotal(20000);
cm.setDefaultMaxPerRoute(10000);
// 关注点 2: 将连接池管理实例配置为 ClientConfig 的属性值
clientConfig.property(ApacheClientProperties.CONNECTION_MANAGER, cm);
// 关注点 3: 配置 Apache 连接器
clientConfig.connectorProvider(new ApacheConnectorProvider());
// 关注点 4: 客户端配置实例携带了连接器信息
client = ClientBuilder.newClient(clientConfig);
```

在这段代码中, 首先定义了连接池管理实例, 该实例通过 setMaxTotal() 方法设置了最大连接数, 通过 setDefaultMaxPerRoute() 方法设置了每路由的默认最大连接数, 见关注点 1。ClientConfig 的属性 ApacheClientProperties.CONNECTION_MANAGER 用于定义连接池管理实例, 其属性值为 PoolingHttpClientConnectionManager 类的实例, 该类是线程安全的, 支持多线程并发操作, 见关注点 2。设置 ClientConfig 的连接器提供者 of ApacheConnectorProvider 实例, 见关注点 3。最后 ClientBuilder 使用携带了连接器信息的 ClientConfig 实例来创建的 REST 客户端实例, 即可实现对 ApacheConnector 连接器的使用, 见关注点 4。

5.3 封装 Client

通常，REST 式的 Web 服务会按模块分别提供独立的 Web 服务，而模块之间的调用通过 Web 服务的 REST API 来实现。因此，每个模块对其他模块的调用是客户端请求，不必在每次请求时重复编写构造客户端实例的代码。为此，抽象出 Client 到公共模块是非常有必要的。封装 Client 可以减少代码冗余，也为统一 Client 构建策略（避免配置不统一带来的混乱）提供了可能，示例代码如下。

```
public T rest(String method, String requestUrl,
    Set<Pair<String, Object>> headParams, Set<Pair<String, Object>> queryParams,
    MediaType requestDataType, Class<T> returnType, T requestData) {
    // 关注点 1: 构造 Client
    if (clientConfig == null) {
        clientConfig = new ClientConfig();
    }
    Client client = ClientBuilder.newClient(clientConfig);

    // 关注点 2: 构造 WebTarget
    WebTarget webTarget = client.target(requestUrl);
    for (Pair<String, Object> queryPair : queryParams) {
        webTarget = webTarget.queryParam(queryPair.getFirst(), queryPair.getSecond());
    }

    // 关注点 3: 构造 Invocation.Builder
    Invocation.Builder invocationBuilder = webTarget.request(requestDataType);
    for (Pair<String, Object> headParam : headParams) {
        invocationBuilder.header(headParam.getFirst(), headParam.getSecond());
    }

    // 关注点 4: 发起请求和结果处理
    javax.ws.rs.core.Response response;
    Entity<T> entity;
    switch (method) {
        case GET:
            response = invocationBuilder.get();
            return response.readEntity(returnType);
        case DELETE:
            response = invocationBuilder.delete();
            return response.readEntity(returnType);
        case PUT:
            entity = Entity.entity(requestData, requestDataType);
            response = invocationBuilder.put(entity);
            return response.readEntity(returnType);
        case POST:
            entity = Entity.entity(requestData, requestDataType);
            response = invocationBuilder.post(entity);
            return response.readEntity(returnType);
        default:
            return null;
    }
}
```

在这段代码中，rest() 方法封装了 HTTP 请求，参数依次是 HTTP 请求方法名、请

求资源地址、请求头参数, 请求查询参数、请求媒体类型、返回值类型、请求实体信息。`rest()` 方法体可以分为 4 个部分, 第一部分是构造 `Client`, 见关注点 1。第二部分是构造 `WebTarget`, 见关注点 2。第三部分是构造 `Invocation.Builder`, 见关注点 3。最后一部分是发起请求和结果处理, 见关注点 4。

5.4 请求 Spring Boot 微服务

5.4.1 不同的 JSON 解析方式

Jersey 集成 Spring Boot (详见 7.2 节) 是目前开发微服务应用的事实标准。但 `spring-boot-starter-jersey` 默认使用 `jackson` 解析 JSON, 而 Jersey 默认使用 `MOXy` 解析 JSON (见 2.3.6 节)。当 Jersey Client 向 Spring Boot 服务请求资源时, 这个差异会导致服务端和客户端对 POJO 的转换不同, 造成反序列化错误, 示例如下。

默认情况下, Jersey 使用 `moxy` 解析, JSON 格式示例如下。

```
{
  "bookList": [
    {
      "book": [
        {
          "bookId": 1,
          "bookName": "JSF2 和 RichFaces4 使用指南 "
        },
        {
          "bookId": 2,
          "bookName": "Java Restful Web Services 实战 "
        },
        {
          "bookId": 3,
          "bookName": "Java EE 7 精髓 "
        }
      ]
    }
  ]
}
```

采用 `jackson` 解析的 JSON 格式示例如下。

```
{
  "bookList" : [ {
    "bookId" : 1,
    "bookName" : "JSF2 和 RichFaces4 使用指南 ",
    "publisher" : null
  }, {
    "bookId" : 2,
    "bookName" : "Java Restful Web Services 实战 ",
    "publisher" : null
  }, {
```

```

    "bookId" : 3,
    "bookName" : "Java EE 7 精髓",
    "publisher" : null
  } ]
}

```

jackson 的解析方式，默认输出全部 field（无论是否为 null）。

对应的 POJO 如下。

```

public class Book {
    private Long bookId;
    private String bookName;
    private String publisher;
}

...

public class Books {
    private List<Book> bookList;

    public Books() {
    }

    public Books(List<Book> bookList) {
        this.bookList = bookList;
    }

    public List<Book> getBookList() {
        return this.bookList;
    }
}
...

```

因此，使用 Jersey Client 访问 Spring Boot 服务时，应顺应其解析方式，采用 jackson 解析 JSON。

首先引入依赖的 jar 包。

```

<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>${jersey.version}</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>${jersey.version}</version>
</dependency>

```

然后在 Client 的 Config 实例中注册 jackson 特性。

```

clientConfig.register(JacksonFeature.class);
...
return ClientBuilder.newClient(clientConfig);

```


5.4.2 完整示例

1. 服务器端

服务器端使用 `@Produces` 注解提供 JSON 格式的媒体格式，不做其他改动。

```
@GET
@Path("books")
@Produces(MediaType.APPLICATION_JSON)
public List<Book> retrieveBooks(@QueryParam("book") final String value) {
    return bookService.retrieve(value);
}
```

2. 客户端

客户端使用 `JacksonFeature` 修改 JSON 解析方式为 `jackson`。

```
public void process() {
    ClientConfig clientConfig = new ClientConfig();
    clientConfig.register(JacksonFeature.class);

    Client client = ClientBuilder.newClient(clientConfig);
    WebTarget webTarget = client.target("http://localhost:8200").
    path("books").queryParam("book", "Java");
    Invocation.Builder invocation = webTarget.request();
    GenericType<List<Book>> genericType = new GenericType<List<Book>>() {};
    List<Book> books = invocation.get(genericType);
    for (Book book : books) {
        System.out.println(book);
    }
}
```

5.5 JavaScript 客户端

除了使用 `Jersey Client` 作为 REST 服务的客户端，还有一类常用的客户端类型就是基于 HTML 的 Ajax 请求。本节将介绍其中两种常用的 JavaScript 技术 `jQuery` 和 `AngularJS` 的 RESTful 请求示例。本节使用的服务端代码见 7.2 节的示例。

首先，我们来看服务端的 WADL。

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <resources base="http://localhost:8080/">
    <resource path="hi">
      <method id="hi" name="GET">
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

这里只有一个服务，即资源路径为 hi 的 GET 方法，表述类型为 JSON。

我们通过 httpie 请求该服务，得到的结果如下所示。

```
http :8080/hi --body
[
  "hello spring boot",
  "hello micro services"
]
```

接下来，我们分别使用 jQuery 和 AngularJS 编码，请求该服务并处理 JSON 结果的展示。

5.5.1 jQuery 客户端

jQuery 是最常用的 AJAX JavaScript 框架，其 API 风格非常先进和易用。首先我们来看 jQuery 请求 RESTful 服务的代码示例。

```
jQuery.noConflict();
jQuery(document).ready(function() {
  jQuery.ajax({
    url: "http://localhost:8080/hi"
  }).then(function(data) {
    jQuery(data).each(function(i, val) {
      jQuery('.hi-content').append(i+" "+val+"<br/>");
    });
  });
});
```

在这段代码中，jQuery.noConflict() 的作用是让渡 \$ 符号给其他 JavaScript 框架，转而使用 jQuery 代替 \$。ajax 方法的入参是 RESTful 资源地址，then 方法是 Promise 模式的结果处理方法，后者的入参为前者的出参，即 JSON 格式的结果数据。jQuery('.hi-content') 是一个 jQuery 类选择器，用于提取 class="hi-content" 的页面元素。

接下来我们看对应的页面代码。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hi jQuery</title>
    <script src="http://libs.baidu.com/jquery/1.10.2/jquery.min.js"></script>
    <script src="hi.js"></script>
  </head>
  <body>
    <div>
      <p>Hi content is:</p>
      <p class="hi-content"></p>
    </div>
  </body>
</html>
```

在这段代码中，引入了 2 个 JavaScript 文件，一个是 jquery1.10.2 的压缩包，一个是上述的 JavaScript 代码文件。在 body 部分，给出了上述 JavaScript 方法所需要的页面元素 `<p class="hi-content"></p>`，用于渲染 JSON 数据。

5.5.2 AngularJs 客户端

AngularJs 是完整的前端数据模型和模板框架，可以实现单网页应用和响应式设计。

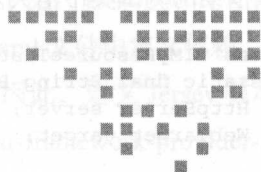
```
function Hello($scope, $http) {
    $http.get('http://localhost:8080/hi').
        success(function(data0) {
            $scope.data = data0;
        });
}
```

在这段代码中，Hello 是 ng-controller 的调用方法，用以完成 HTTP GET 请求，回调函数中将结果赋值给 \$scope.data，页面中可以通过 {{data}} 获取该值。HTML 代码如下。

```
<!doctype html>
<html ng-app>
  <head>
    <title>Hello AngularJS</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.8/
angular.min.js"></script>
    <script src="hello.js"></script>
  </head>
  <body>
    <div ng-controller="Hello">
      <p>Hi content is:</p>
      <p>{{data}}</p>
    </div>
  </body>
</html>
```

5.6 本章小结

本章讲述了 JAX-RS2 定义的 REST 客户端及其初始化。客户端的实现需要注意的是客户端的实例数量不宜过多，可以根据情况使用 5.2 节讲述的连接器来避免每次请求实例化一个客户端的情况，5.3 节演示了 REST 客户端的初始化代码块的封装和复用，5.4 节对 Spring Boot 和 Jersey 对 JSON 的不同处理方式进行了分析并提供了解决方案，5.5 节给出 2 个纯 HTML 的 RESTful 客户端示例。



第 6 章

Chapter 6

REST 测试

本章讲述如何使用 Jersey 提供的测试框架实现对基于 JAX-RS2 的 Web 服务进行自动化测试。

自动化测试是软件质量保证的必要手段，是解放开发团队于重复劳动的法宝。同时也是敏捷开发的重要环节，项目源代码的代码覆盖率即指测试代码中对源代码的公有方法的覆盖情况，是 CI（Continuous Integration，持续集成）和 CD（Continuous Delivery，持续交付）中的重要指标之一。

使用 JUnit 结合 Jersey 的测试框架，能够轻松地实现单元测试、CI 测试和系统测试。Jersey 提供了 4 种内置容器，测试过程对外置容器没有天然的依赖。

阅读指南

本章示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/6.rest-test>。

6.1 Jersey 测试框架

Jersey 核心测试框架包 jersey-test-framework-core 提供了测试环境中容器和客户端示例，使开发者不必编写管理容器和客户端的样板代码。

在没有 Jersey 核心测试框架支持的测试类中，开发者需要在编写测试用例的同时，维护测试容器和客户端代码。setUp 方法中的代码都是样板代码，这些样板代码会在使

Jersey 核心测试框架的测试类中消失。没有使用 Jersey 测试框架的测试，示例代码如下。

```
public class TIMyResourceTest {
    public static final String BASE_URI = "http://localhost:8080/webapi/";
    private HttpServer server;
    private WebTarget target;

    @Before
    public void setUp() throws Exception {
        final ResourceConfig rc = new ResourceConfig().packages("com.example");
        final URI uri = URI.create(BASE_URI);
        server = GrizzlyHttpServerFactory.createHttpServer(uri, rc);
        server.start();
        final ClientConfig cc = new ClientConfig();
        final Client client = ClientBuilder.newClient(cc);
        target = client.target(BASE_URI).path("books");
    }

    @After
    public void tearDown() throws Exception {
        if (server != null) {
            server.stop();
        }
    }

    @Test
    public void testQueryGetXML() {
        final WebTarget queryTarget = target.path("/book").queryParam("id", Integer.
            valueOf(1));
        final Invocation.Builder invocationBuilder = queryTarget.request(MediaType.
            APPLICATION_XML_TYPE);
        final Response response = invocationBuilder.get();
        ...
    }
}
```

使用了 Jersey 核心测试框架后，代码量减少了很多，示例代码片段如下。

```
public class TIMyResourceJTFTTest extends JerseyTest {
    private static final String BASEURI = "books/";
    @Override
    protected Application configure() {
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testQueryGetXML() {
        final WebTarget queryTarget = target(BASEURI + "book").queryParam("id",
            Integer.valueOf(1));
        final Invocation.Builder invocationBuilder = queryTarget.request(MediaType.
            APPLICATION_XML_TYPE);
        final Response response = invocationBuilder.get();
        ...
    }
}
```

在这段代码中，TIMyResourceJTFTTest 类继承了 Jersey 核心测试框架类 JerseyTest，其测试代码中只有覆盖 configure 方法是与测试用例无关的代码。在测试用例中可以直接使用

target 方法从测试框架中获取 WebTarget 类的实例，余下的代码都是测试用例本身的内容。

在项目中使用 Jersey 核心测试框架包需要在 pom.xml 文件中定义依赖。Jersey 核心测试框架提供了 4 种内置容器，默认使用 Grizzly2 容器。因此，定义 Jersey 核心测试框架依赖的同时，需要定义测试框架 Grizzly2 容器包 jersey-test-framework-provider-grizzly2。否则，运行时会报 TestContainerException 异常。

Jersey 测试框架核心的依赖定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework</groupId>
  <artifactId>jersey-test-framework-core</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

下面是 4 种内置容器的定义。

(1) Jersey 内置容器 Grizzly2 依赖定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架 Grizzly2 容器所在的模块为 jersey-test-framework-provider-grizzly2，Grizzly 轻量级 HTTP 容器是 Jersey 测试框架的默认容器。

(2) Jersey 内置容器 inmemory 依赖定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-inmemory</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架内存容器所在的模块为 jersey-test-framework-provider-inmemory，内存容器不是一个真正的容器。测试框架直接调用应用的 API，因此没有真正的网络通信。内存容器不支持真正的容器特性和 Servlet，但是非常适合作为单元测试阶段的容器。

(3) Jersey 内置容器 JDK-HTTP 依赖定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-jdk-http</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架 JDK 版 HTTP 容器所在的模块为 jersey-test-framework-provider-jdk-http，JDK 版 HTTP 容器是 JDK。

(4) Jersey 内置容器 simple-HTTP 依赖定义如下。

```

<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-simple</artifactId>
  <version>${jersey.version}</version>
</dependency>

```

测试框架简单版 HTTP 容器所在的模块为 jersey-test-framework-provider-simple，集成 Jersey 的轻量级 HTTP 容器。

在测试类中使用 4 种内置容器非常方便，只需要在覆盖 configure 方法中定义容器工厂名称即可。TestProperties.CONTAINER_FACTORY(jersey.config.test.container.factory) 参数的值被设置为 org.glassfish.jersey.test.simple.SimpleTestContainerFactory。

```

public class TMyResourceJTFTTest extends JerseyTest {
    static final String CONTAINER_GRIZZLY = "org.glassfish.jersey.test.grizzly.
    GrizzlyTestContainerFactory";
    static final String CONTAINER_MEMORY = "org.glassfish.jersey.test.inmemory.
    InMemoryTestContainerFactory";
    static final String CONTAINER_JDK = "org.glassfish.jersey.test.jdkhttp.JdkHttp
    ServerTestContainerFactory";
    static final String CONTAINER_SIMPLE = "org.glassfish.jersey.test.simple.
    SimpleTestContainerFactory";
    @Override
    protected Application configure() {
        set(TestProperties.CONTAINER_FACTORY, CONTAINER_SIMPLE);
        return new ResourceConfig(BookResource.class);
    }
}

```

6.2 单元测试

单元测试用一句话概括就是对源代码中公有方法的功能性测试。本节不会对单元测试的基本概念和测试方法进行详述，着重讲述单元测试的难点。

6.2.1 集成 Spring 的单元测试

第一个难点是如何在单元测试中使用 Spring 容器。Spring 提供了相关的解决方案，可以在测试类的注解中指定 SpringJUnit4ClassRunner 来实现。请看下面的详述。

1. 依赖注入

对于使用依赖注入的源代码，单元测试最常见的问题是被测试类 A 依赖注入了 B 类，运行时 B 的实例总是空指针。这个问题是因为单元测试的上下文中没有依赖注入容器的环境。以 Spring 作为依赖注入容器的项目为例，BookService 类中依赖注入了 BookDao 类的实例，那么测试代码中，BookService 类的实例就不能直接通过构造函数创建，否则这个 BookService 实例就不是 Spring 容器管理的 Bean，导致的结果就是 BookDao 实例为空指针，

示例代码如下。

```
public class BookService {
    @Autowired
    private BookDao bookDao;
    public Book saveBook (final Book book) {
        return bookDao.store (book);
    }
}
```

这种情况下，我们可以在测试类中使用 `RunWith` 注解定义 `SpringJUnit4ClassRunner` 类。与 Jersey 测试框架帮我们管理测试服务器和客户端类似，`spring-test` 包提供的 `SpringJUnit4ClassRunner` 类可以帮我们管理 Spring 容器。通过 `ContextConfiguration` 注解定义的 Spring 配置文件，`SpringJUnit4ClassRunner` 实例在测试类启动后，帮我们建立了 Spring 容器环境，这样就可以让容器帮我们生成 `BookService` 类的实例，这一过程中，`BookDao` 类的实例也创建好并存在于 Spring 容器中了。单元测试代码使用 `@Autowired` 定义了一个 `BookService` 字段，其余部分只和测试用例有关，示例代码如下。

```
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class TUMyServiceTest {
    @Autowired
    private BookService bookService;
    @Test
    public void testGetAndSave() {
        final Book result = bookService.getBook(1L);
    }
}
```

2. 事务

在使用 JPA 的项目中以及对 DAO 层源代码的测试中，一个常见的“怪现象”是明明存储一个实体类已经通过，数据库中却没有这条记录。这是因为 JPA 需要显式地提交事务才会将数据真正写入数据库。在集成 Spring 的项目中，通常使用 `@Transactional` 注解存储方法，即可在运行时由 Spring 容器提交事务，如下所示。

```
@Repository
public class BookDao {
    @Transactional
    public Book store (final Book entity) {
        return entityManager.merge (entity);
    }
}
```

但是，项目中的业务逻辑不总会允许每一个实体的存储都原子地提交事务，而是当 A、B、C 都成功保存后才会提交事务，否则回滚事务。因此，`@Transactional` 注解会从 DAO 层移至 Service 层。这种情况下，对 DAO 层的单元测试再次陷入到“怪现象”的窘境里。诚然，有两种情况可以忽略测试数据的持久化，一种是使用内存数据库（通常，单元测试应该这样做），另一种是测试逻辑成功即可，测试数据没有意义。如果存储测试数据是我们的单

元测试所必须的，只有在测试类中寻找实现事务显式提交的办法。这种情况下，我们可以使用 `TransactionConfiguration` 注解和 `Transactional` 注解，如下所示。

```
@ContextConfiguration(locations = { "classpath:context.xml", "classpath:
context-persistence.xml", "classpath:context-transaction.xml" })
@Transactional(transactionManager="xxxTransactionManager",defaultRol
lback=false)
@Transactional
@RunWith(SpringJUnit4ClassRunner.class)
public class TITestXxxDao {
```

6.2.2 异步测试

单元测试中另一个难点是对实现异步功能的源代码进行测试。一个常见的现象是被测试的线程还在进行中，而测试主线程已经结束。JUnit 框架同步测试的扩展可以从编码和架构上解决。

从编码上解决异步使用第 8 章讲述的 Java 并发技术。比如，在主线程中使用 `Future` 和同步器来阻塞主线程，直到被测试的线程结束（代码的实现请参考第 8 章示例）。也可以在测试用例中创建线程或使用线程池来测试异步程序。异步测试代码的编写相对难度大些，但可以快速解决对功能模块的单元测试，需要读者仔细揣摩。这里要说明的是，异步单测用例通常比较耗时，因此在全量执行单元测试时应当排除这部分用例。一个通用的做法是使用 Maven 定制 profile 来隔离不同的测试用例。

架构上可以选择 JUnit 的第三方扩展包来支持多线程测试。这超出了本书的范围，请读者参考 JUnit 扩展的相关资料。从实际开发和单测经验上看，架构层面的解决方案更重量级，适合团队规定使用的场景；编码解决更轻量 and 灵活，但不具有通用性。

阅读指南

异步测试的示例请参考第 4 章的示例项目 <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/4.2.asynchronized>。

6.3 集成测试

集成测试是系统运行时的功能测试，是对组件、模块之间联合工作的测试。REST 应用的集成测试就是对系统公开的资源地址的功能性测试。运行时测试依赖于测试服务器。根据笔者经验，测试服务器的建立与启动可以分为 3 类。

第一类是使用如上所述的内置 HTTP 容器。这种方式通过 JUnit 测试类直接定义测试服

务器或者继承 Jersey 测试框架类 `JerseyTest`，使其在执行测试前加载配置并启动服务，在测试用例完成后停止服务。

第二类是使用 Maven 插件启动外置服务器。这种方式通过在项目的 Maven 配置文件 `pom.xml` 中定义测试服务器，使其在 Maven 生命周期的 `pre-integration-test` 阶段启动，并在 `post-integration-test` 阶段停止。参考第 2 章关于 Servlet 容器应用一节。

以上两种方式都无须部署 (deploy) 到测试服务器。

第三类是将项目打包部署到和生产环境相同的集成测试环境。其优点是可以模拟真实场景，以完成对系统功能性和非功能性的测试。缺点是打包部署占用了集成测试的时间。典型的 Servlet 容器是 Tomcat，Java EE 容器是 GlassFish，参考第 1 章关于 Servlet 容器应用一节。

6.4 日志增强

在测试日志中，Jersey 测试框架提供了输出通信过程中的传输日志信息。在测试代码中启用这一功能是通过覆盖 `JerseyTest` 类的 `configure` 方法并加入关注点 1 一行，示例代码如下。

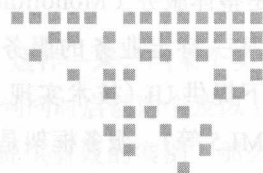
```
public class TIMyResourceJTFTTest extends JerseyTest {
    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC); // 关注点 1 启用传输日志
        enable(TestProperties.DUMP_ENTITY);
        //set(TestProperties.CONTAINER_FACTORY, CONTAINER_SIMPLE);
        return new ResourceConfig(BookResource.class);
    }
}
```

这样一来，测试过程中输出的信息将包括传输日志，请参考下面的输出。在启用传输日志后，可以启用实体日志来输出传输过程中的实体信息，请参见下面最后一段所示的 XML 格式的数据。

```
1 > GET http://localhost:9998/books/
2 < 200
2 < Date: Tue, 22 Oct 2013 03:24:46 GMT
2 < Content-Length: 496
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><books><bookList><book bookId="1" bookName="Java Restful Web Service 使用指南"
publisher="cmpbook"/><book bookId="2" bookName="JSF2 和 RichFaces4 使用指南"
publisher="phei"/></bookList></books>
```


6.5 本章小结

本章详细讲述了 Jersey 的测试框架及其在单元测试和集成测试中的使用。单元测试主要考虑依赖注入和事务这两个技术点如何和生产环境保持一致。集成测试主要考虑如何部署测试环境，使之尽量贴近生产环境。最后介绍了 Jersey 测试框架对日志信息的增强。下一章，我们开始讲述 REST 的服务器端推送技术和异步通信。



第 7 章

Chapter 7

微 服 务

微服务 (Micro Service) 是由马丁·福勒 (Martin Fowler) 于 2014 年提出的一种架构风格。

和 RESTful 一样，微服务没有因其概念的提出而诞生一件新的技术，却同样对行业影响深远。相对于整体风格 (Monolithic style) 和面向服务架构 (SOA)，微服务的设计更加轻盈、灵活，开发周期急剧缩短，部署非常容易且有很高的伸缩性。微服务架构更能被敏捷团队、DevOps 所接受，这背后的道理，康威在 1967 年已经总结 (Canway's law)：设计系统的组织，最终会将其沟通结构复制到产品的设计中。

传统软件公司的组织结构，通常包括 (产品) 需求、架构设计、平台开发、业务开发、UI 开发、测试以及 DBA 等角色。产品会按业务模块划分，由架构设计主导实现需求，平台开发只关心如何增加平台功能、改进组件或者中间件的功能和性能，业务开发只关心如何实现具体业务，测试则对需求的质量负责。这种传统的瀑布模型忠实地按照整体风格实现着一版又一版的需求，同时也在修复着每版带来的缺陷。

马丁·福勒强调，组织要面向服务。将原有的组织分层 (UI/Server/DBA) 推翻，建立了微服务下的组织分层 (Orders/Shipping/Catalog)，组织提供 smart endpoint 使服务直达用户。Google、Amazon、eBay 以及 Netflix 等公司与之不谋而合，积极倡导并履行着将测试、运维的角色抹平，开发人员对自己代码的质量、部署和运维负责。开发者要站出来应对构建生命周期的全部，比如从自动化测试到一键发布，从配置脚本到金丝雀部署等。Thoughtworks 还为此提出了持续交付的完整流水线。开发人员由此成为构建生命周期的全栈工程师。

微服务不是整体服务 (Monolithic Service) 中的一个组件, 而是以业务领域为中心, 高内聚的、实现某一特定业务的服务。微服务通常以 HTTP+JSON 的形式开放 RESTful 接口, 某些情况下提供 UI (技术实现上选择很多, 比如纯 HTML 技术 Bootstrap+jQuery、单网页技术、HTML5 等)。服务框架是微服务的重点, 在 Java 领域, 通常皈依在 Spring Boot 门下 (这也是本章我们要说的重点)。服务设计上, 一如当年为了设计好面向对象, 大家要反复思考著名的 23 个设计模式一样, 今时要设计好微服务, 也需要认真研读领域驱动设计 (Domain-Driven Design, DDD)、企业应用架构模式 (Patterns of Enterprise Application Architecture) 等著作。服务通信上, JVM 圈的典范是 Netty 和 AKKA, 如今, 事件驱动的异步 IO 已经是标配, 如果能做到没有竞态则是最好。以此通信为基础, 服务自身与消息系统、存储系统的交流自不在话下。开发人员由此成为服务技术栈的全栈工程师。

交织在构建与实现之间, 开发人员成为服务产品可用性的负责人。于是, 开发人员便由此不断与产品经理沟通产品的未来、与其他服务的产品经理沟通如何对接, 他们要遵循伯斯塔尔法则 (Postel's law), 设计服务的 API 像设计 UNIX 管道一样, 宽进严出。优秀的开发人员或许在思考哲学的根本问题——我是谁。因为做到如上全部, 已经不是一个传统意义上的开发人员了。

组织和人的进化成为微服务孕育而生的客观条件, 微服务的发展又反作用于人的综合发展。

本书的读者在微服务上已经占得先机, 那就是具备设计和实现 HTTP+JSON 服务接口的能力。那么, 以此为起点, 如何才能掌握微服务技术栈呢? 首先是向下的能力, 除去设计思想这些依赖于思考、视野和经验等因素, 从实践角度上说我们需要 Spring Boot 帮助我们快速开发、快速部署, 需要 Spring Cloud 帮助我们确保系统的高可用, 需要 Docker 等容器技术帮助我们实现弹性部署和运维。这些技术也许应对微服务还不够全面, 但在我看来, 它们是微服务主要的技术基础。本章将分别讲述 Spring Boot 和 Spring Cloud, 第 8 章将讲述 Docker 等容器技术。

7.1 微服务技术栈

本节将向读者总体介绍实现微服务应用的技术栈, 后面两节是具体的实战内容, 实战部分集中体现了如何让服务获得高可用和可伸缩性。

通过之前章节的阅读和实战, 我们有了 war, 当将其部署在 Tomcat、Jetty 等容器上后, 我们便有了一个微服务 (本书不涉及微服务的设计思想, 只关注技术实现)。那么第一个问题来了, 如何做到服务可用呢?

7.1.1 服务发现

为了确保服务的可用性，我们可以部署多个实例。这样一来，当某个实例出现故障时，我们还有其他实例可以提供服务（当然，你的服务应做到同时启动两个及以上的实例时，彼此之间没有逻辑问题）。为此，我们要为服务的使用者提供有效的实例。那么，谁来负责告诉使用者哪些实例是有效的呢，在有效的实例中要让使用者用其中的哪一个呢？

1. 健康检测

对于实例的有效性，我们需要一种可以知道各个实例是否健康的机制，它被称为健康检测（Health Indicator）或者故障检测（Failure Detection）。原理是，在实例启动时将自己注册到协调系统中，协调系统通过心跳机制（heartbeat）定期询问每个实例，或者实例自己上报。如果请求直接失败或者经过一段时间（即 timeout 定义的时间）后实例没有响应，即认为该实例故障。故障后，协调系统可以选择将其注销，过段时间再询问该实例等策略。

协调系统自身也是高可用的，通常是指分布式协调系统，其中的典范是 Apache Zookeeper(<http://zookeeper.apache.org>)，开源领域的 Google Chubby。后者主要有与 Vagrant 同出一门的 HashiCorp Consul(<https://www.consul.io>)，以及 CoreOS Etcd(<https://github.com/coreos/etcd>)。

2. 路由及 API 代理

有了协调系统，我们就可以使用定时更新程序从协调系统上获取健康实例列表，并将其反馈给路由系统。负责告诉使用者哪些实例就是站在最前面的路由，或者叫反向代理（Reverse Proxy），如果它在整个微服务数据中心负责路由多种微服务，那么它又叫作 API 代理（API Proxy）。

对于让使用者用哪个实例，要比前面一个问题简单。最直接的方案是让路由采用轮询策略，按顺序返回每个健康的实例，这也是负载均衡（Load Balance）中最常用的方式。如果服务部署在计算能力不同的资源环境中时，可以考虑加权的路由策略。

支持这一解决方案的前置代理服务有 Nginx、HAProxy、Vulcand 等，定时更新配置的服务有 Confd、Consul Template 等。

7.1.2 可伸缩性

有了服务发现，实例的数量就可以动态调配了。这可以解决很多场景中的问题。

1. 资源不足

假设我们初始启动了某一服务的 3 个实例（A、B、C），在运行一段时间后，由于压力过载，A 实例失去响应或者宕掉，我们可以追加 3 个新的实例（D、E、F）上去，此时有效

的服务实例为：B ~ F，5 个实例在一定程度上能有效地分担负载。监控和动态管理服务实例的工作可以使用自动化运维工具来完成，也可以结合容器技术实现。

2. 并行计算

如果我们提供的服务是用于计算的，那么分片 + 路由的并行计算方案能有效节省时间。比如有 10 亿量的数据清洗任务，如果启动 5 个实例，每实例处理的数据分片就降到 2 亿，如果增加到 100 个实例，每实例处理的数据量就只有 1 千万。亦或是，将服务用于并行处理持续集成中的构建、测试、度量、报告等任务。

3. 资源调配

上述情况也许不会一直持续，服务总会回到没那么繁忙的时候。此时，之前启动的那么多多个实例就浪费了。合理的资源分配方案是缩减服务实例数量，将服务器资源让渡给其他（用户的）繁忙的服务去创建实例。

7.1.3 回到起点

具备了可伸缩性，我们的微服务可以称为一个产品了。那么，上述的这些技术点我们该如何各个击破呢？又当如何将它们穿在一起呢？让我们再次回到起点。

1. Spring Boot

首先，我们需要 Spring Boot 这样的快速创建 REST 服务并快速部署的框架。

传统的 war+Tomcat 在今天来说还是太重，我们很难实现自动化运维。可能有读者想到了使用 Maven 插件启动和停止 Tomcat 上的服务，那么我们为什么不用 Maven 插件直接部署和停止应用本身呢？对于分布式部署或者容器化来说，Tomcat 也是个累赘。Spring Boot 可以把容器、服务所有的依赖以及服务自身打成一个 jar 包，我们直接运行 jar 便部署了一个服务。如果我们把 jar 放到容器里边，便有了容器化的微服务，就这么简单。

Spring Boot 可以把 Spring 的一切都纳入进来，比如 Spring MVC、Spring Data、Spring Batch、Spring Security 等，如果我们的服务需要，这种无缝集成会让开发变得惬意。

2. Spring Cloud

Spring Boot 提供了服务化的能力，Spring Cloud 提供了在各种云平台服务化的能力。

我们引入 Spring Cloud 并非要讲如何与各种云集成，而是从更通用的角度，讲述它的 3 个出色实现了服务发现能力的子项目：Spring Cloud Zookeeper、Spring Cloud Consul、Spring Cloud Etcd，具体参见 7.3 节。

7.2 REST 服务与 Spring Boot

Spring Boot 的出现, 让基于 Spring 框架的开发、配置和部署变得更加方便, 从 0 到 1 的时间急剧减少。默认情况下, Spring Boot 内置了 Tomcat 容器, 通过切换 starter 将内置 Servlet 容器更换为 Jetty 或者 Undertow (JBoss Wildfly 的 Web 服务器), 从此可以告别 war。Spring Boot Actuator 提供了监控和度量能力, 对于全栈工程师来说, 除了窃喜还有什么呢?

Spring Boot 项目的官方网站地址为 <http://projects.spring.io/spring-boot>, Spring Boot 源代码地址为 <https://github.com/spring-projects/spring-boot>。其中包括了 Spring Boot 的全部示例, 路径为 `spring-boot-samples`。

阅读指南

7.2 节的示例源代码地址为: <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/7.2.demo>。

7.2.1 Bootiful

Bootiful 是 Josh Long 在 SpringOne 2015 大会上提出的, 意指使用 Spring Boot 快速开始各种类型服务的开发和配置。使用者可以通过 <http://start.spring.io> 网站来选择工程的依赖并设定工程属性, 如图 7-1 所示。

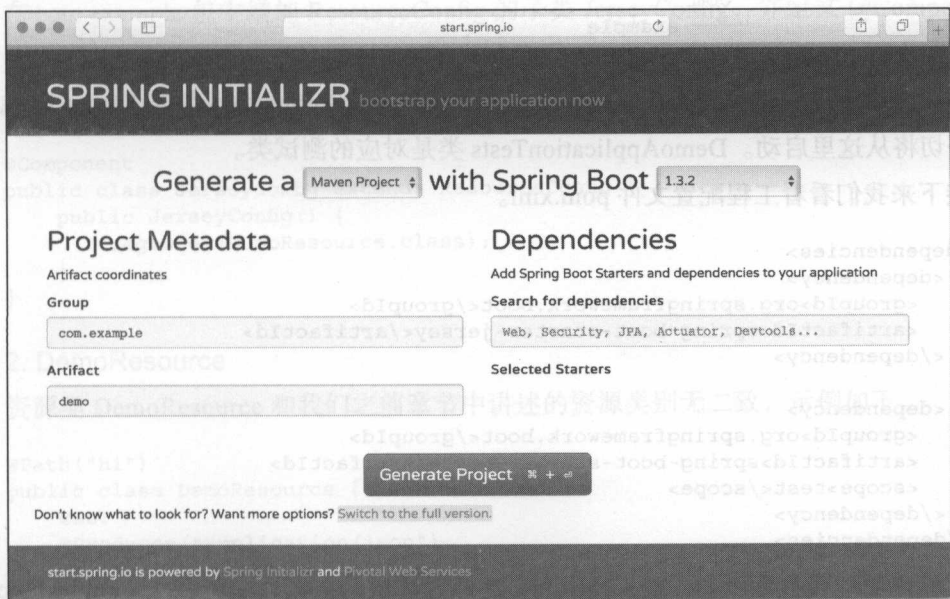


图 7-1 通过 Spring Initializer 页面实现 Bootiful

在图 7-1 中，我们可以选择工程的构建工具（Maven 或者 Gradle）和 Spring Boot 的版本号。工程的元数据对应构建配置中的坐标、组和工件。工程的依赖是基于构建工具的配置信息，这些被依赖的 Spring 组件以 starter 的形式加入工程。如果想知道 Spring Initializer 提供了哪些依赖，可以单击页面下方的 Switch to the full version 链接，切换到完整的 Starts 视图。

我们通过这个页面开始第一个实战。在依赖中填入 Jersey (JAX-RS)，其他选项都使用默认的，然后单击下方的生成工程按钮，我们将得到一个下载到本地的工程压缩包 demo。

1. 工程结构

进入 demo 目录，通过 tree 命令查看工程结构如下。

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── example
│   │   │           └── DemoApplication.java
│   │   └── resources
│   │       └── application.properties
│   └── test
│       ├── java
│       │   └── com
│       │       └── example
│       │           └── DemoApplicationTests.java
```

这是典型的 Maven 工程结构，源代码目录的 DemoApplication 类是 Spring Boot 的主类，一切将从这里启动。DemoApplicationTests 类是对应的测试类。

接下来我们看看工程配置文件 pom.xml。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

依赖声明中包含两个 starter，其中 test 是单元测试的基础，它依赖了 junit 和 mockito。jersey 是我们在页面上选定的，它的主要依赖包括：内置 Tomcat 包 spring-boot-starter-

tomcat、Spring Web 4.2.4 包 spring-web、Jersey 2.22.1 服务端包 jersey-server、jersey-container-servlet-core、jersey-container-servlet 以及 Jersey 和 Spring 集成包 jersey-spring3(需要注意的是,它依赖的 JSON 解析器是 jersey-media-json-jackson,而不是 Jersey 服务器默认使用的 jersey-media-json-moxy)。

pom.xml 中还声明了一个构建插件 spring-boot-maven-plugin,该插件的作用是将工程构建为可以就地运行的 jar。

mvnw 是 Gradle 风格的执行脚本,目的是在没有 Maven 环境的情况下,执行构建。如果本地有符合 Spring Boot 构建要求的 Maven 版本,可以忽略该脚本。

2. 配置

resources 下的 application.properties 文件是 Spring Boot 的默认配置文件。作为示例,我们添加如下行,重新定义服务的端口(默认为 8080)。

```
server.port=18080
```

7.2.2 RESTful

我们现在有了 Spring Boot 框架,只需要增加如下两个类,即可完成 RESTful 服务的实现。

1. JerseyConfig

在 com.example 包中增加 ResourceConfig 的子类 JerseyConfig,并定义 @Component 注解。这样在 Spring 启动时,会自动加载 JerseyConfig,在 JerseyConfig 构造子中将资源类 DemoResource 注册到 Spring 上下文。

```
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        register(DemoResource.class);
    }
}
```

2. DemoResource

资源类 DemoResource 和我们之前章节中讲述的资源类别无二致,示例如下。

```
@Path("hi")
public class DemoResource {
    @GET
    @Produces("application/json")
    public List<String> hi() {
        List<String> result = new ArrayList<>();
        result.add("hello spring boot");
    }
}
```

```

        result.add("hello micro services");
        return result;
    }
}

```

3. 运行和测试

完成了代码编写后，我们可以通过 `mvn spring-boot:run` 命令原地启动服务。

使用 `cURL` 测试 REST 服务。

```

curl :18080/hi

["hello spring boot","hello micro services"]

```

通过 `Jersey Client` 编写测试代码。

```

@Test
public void testHi() {
    Client client = ClientBuilder.newClient();
    WebTarget webTarget = client.target("http://localhost:18080/hi");
    List list = webTarget.request(MediaType.APPLICATION_JSON_TYPE).get(List.class);
    Assert.assertNotNull(list);
    Assert.assertEquals("hello spring boot", list.get(0));
}

```

4. 启动和停止

使用 `mvn clean install -DskipTests` 命令构建工程，然后将 `target` 目录下的 `jar` 拷贝到服务器端，运行 `nohup java -jar demo-0.0.1-SNAPSHOT.jar &` 就完成工程部署了。

如果要停止服务，可以通过 `ps -aux | grep java` 查找 `java` 进程，并获得 `demo` 服务的 `PID`，然后 `kill` 掉这个进程。

另一个办法是在启动后将进程号存入文件，在停止时 `kill` 改进程号。示例如下。

```

$ nohup java -jar demo-0.0.1-SNAPSHOT.jar --server.port=18081 &
$ echo $! > demo@18081.pid

$ curl :18081/hi
["hello spring boot","hello micro services"]

$ kill -9 $(cat demo@18081.pid)
[1]  + 9835 killed nohup java -jar demo-0.0.1-SNAPSHOT.jar --server.port=18081

```

7.2.3 Actuator

`Spring Boot Actuator` 为开发人员提供了监控、统计、度量等运维功能，也为性能调优带来方便。

基于上述的 `demo` 工程，我们增加两个依赖。

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

我们看到增加了两个 starter，为了启用 Actuator，我们增加了 actuator，那么为什么要增加 web 呢？这是因为单纯的 Jersey 工程无法支持 Actuator，必须是 Spring MVC 才能启用 Actuator。

在 Jersey 中引入 Spring MVC 会带来根路径冲突的问题，因为它们各自的 Servlet 都默认处理根路径。一种办法是通过如下配置，分别指定各自的根路径。

```

# Spring MVC
server.servlet-path=/sys
# Jersey
spring.jersey.application-path=/rest

```

但是，上面这样配置会影响已有 API，当然我们也可以只改其中的一个，但只是为了 Actuator，这么做不够优雅。一个比较巧妙的办法是让 Spring MVC 只处理 Actuator 的事情，Jersey 专心处理 REST 业务。具体做法是分别指定服务端口。

```

server.port=${port:8080}
management.port=${mport:8081}

```

这样一来，虽然两者还是都处理根路径，但端口号不同，彼此就没有冲突了。上面这样定义还有另外一层意思，就是为默认的配置项提供了一个别名，我们启动服务时，可以使用别名来指定该配置项的值。

```
nohup java -jar target/demo-0.0.1-SNAPSHOT.jar --port=1234 --mport=1314 &
```

使用 httpie 测试服务是否可用。

```

$ http --body :1234/hi
[
  "hello spring boot",
  "hello micro services"
]

```

1. 健康检查

有了 Actuator，服务可用性检测可以请求专门用于健康检查的端点。

```

$ http --body :1314/health
{
  "diskSpace": {
    "free": 9319469056,

```



```

        "status": "UP",
        "threshold": 10485760,
        "total": 499048775680
    },
    "status": "UP"
}

```

2. 访问追踪

服务启动后，执行了上述两个请求，现在我们来查看访问打点日志。

```

$ curl :1314/trace
{"timestamp":1455981388422,"info":{"method":"GET","path":"/hi","headers":
{"request":{"host":"localhost:1234","connection":"keep-alive","accept-
encoding":"gzip, deflate","accept":"*/*","user-agent":"HTTPie/0.9.3"},"response
":{"X-Application-Context":"application:1234","status":"200"}}}}

```

从请求的响应可以看到，trace 只记录了 REST 服务端口 1234 上的请求，而没有把监控本身的请求纳入进来，这正是我们希望的结果。默认情况下，trace 会缓存最后 100 个请求事件。

3. 配置信息

我们见识了 Actuator 的优势，那么 Actuator 还提供了哪些端点呢？我们试试配置属性端点 configprops，看看是否可以获得有用的信息。

```

$ http --pretty all ':1314/configprops' | grep prefix | grep endpoints
"prefix": "endpoints.autoconfig",
"prefix": "endpoints.beans",
"prefix": "endpoints.configprops",
"prefix": "endpoints.dump",
"prefix": "endpoints",
"prefix": "endpoints.jmx",
"prefix": "endpoints.env",
"prefix": "endpoints.health",
"prefix": "endpoints.info",
"prefix": "endpoints.metrics",
"prefix": "endpoints.mappings",
"prefix": "endpoints.shutdown",
"prefix": "endpoints.trace",

```

从请求的响应中，我们最终可以发现，以 endpoints 开头的是可以用于监控的端点。我们从中摘取两个作为示例。

4. 运行环境信息

运行环境信息是运维阶段很有价值的信息，比如进程号、端口号、路径信息等，示例如下。

```

$ http --body :1314/env
{

```

```

"applicationConfig: [classpath:/application.properties]": {
  "management.port": "${mport:8081}",
  "server.port": "${port:8080}"
},
"commandLineArgs": {
  "mport": "1314",
  "port": "1234"
},
"systemEnvironment": {
  "JAVA_HOME": "/Users/erichan/.jenv/candidates/java/current",
  ...
"systemProperties": {
  "PID": "15416",
  "catalina.home": "/private/var/folders/f5/8bdk7hs93d981hz80g0wq3bw0000gn/T/tomcat.1138336312122745083.1234",
  ...
}

```

5. 度量信息

运行时的度量对了解服务系统的总体状态非常有益，示例如下。

```

http --body :1314/metrics
{
  "classes": 7306,
  "classes.loaded": 7306,
  "classes.unloaded": 0,
  "counter.status.200.hi": 2,
  "gauge.response.hi": 1.0,
  "gc.ps_marksweep.count": 2,
  "gc.ps_marksweep.time": 146,
  "gc.ps_scavenge.count": 8,
  "gc.ps_scavenge.time": 71,
  "heap": 3728384,
  "heap.committed": 1000960,
  "heap.init": 262144,
  "heap.used": 196008,
  "http.sessions.active": 0,
  "http.sessions.max": -1,
  "instance.uptime": 2271019,
  "mem": 1059647,
  "mem.free": 804951,
  "nonheap": 0,
  "nonheap.committed": 60048,
  "nonheap.init": 2496,
  "nonheap.used": 58688,
  "processors": 8,
  "systemload.average": 1.7861328125,
  "threads": 30,
  "threads.daemon": 27,
  "threads.peak": 30,
  "threads.totalStarted": 37,
  "uptime": 2274087
}

```

7.3 REST 服务与 Spring Cloud

Spring Cloud 秉承了 Spring 组件的一贯风格，使用统一的编程方式与不同的服务组件集成。Spring Cloud 是 Spring Boot 在分布式方向上的延伸，使用注解实现配置，使用单一的 jar 包实现部署，使我们开发的服务具备很好的伸缩性和灵活性。同时 Spring Cloud 提供了服务发现、集中式配置，集成了智能路由和负载均衡、断路器等组件。我们的重点是微服务的高可用性，本节将重点关注如何使用 Spring Cloud 实现服务发现。我们将依次接触到 Apache Zookeeper、Nginx、Consul 和 Etc。

7.3.1 Spring Cloud Zookeeper

Spring Cloud Zookeeper 是 Spring Cloud 与 Apache Zookeeper 集成的实现。该项目的地址为：<http://cloud.spring.io/spring-cloud-zookeeper>，源代码地址为：<https://github.com/spring-cloud/spring-cloud-zookeeper>。

1. Zookeeper 服务

开发环境下，Zookeeper 的极简安装和启动示例如下。完整的集群环境详见第 8 章，我们将使用 Docker 来实现。

```
brew install zookeeper

==> Downloading https://homebrew.bintray.com/bottles/zookeeper-3.4.7.el_
capitan.bottle.tar.gz
...
To have launchd start zookeeper at login:
  ln -sfv /usr/local/opt/zookeeper/*.plist ~/Library/LaunchAgents
Then to load zookeeper now:
  launchctl load ~/Library/LaunchAgents/homebrew.mxcl.zookeeper.plist
Or, if you don't want/need launchctl, you can just run:
  zkServer start
==> Summary
/usr/local/Cellar/zookeeper/3.4.7: 235 files, 18M
```

使用 `zkServer start` 启动 Zookeeper 服务，使用 `zkCli -server 127.0.0.1:2181` 命令快速验证服务是否启动。

2. 官方示例

官方示例是了解和熟悉一个项目的最佳入门材料。Spring Cloud Zookeeper 的源代码项目中提供了示例项目 `spring-cloud-zookeeper-sample`。按照 README 的指导，我们启动如下 3 个服务实例。

```
mvn install -DskipTests
```

```
java -jar spring-cloud-zookeeper-sample/target/spring-cloud-zookeeper-sample-1.0.0.BUILD-SNAPSHOT.jar
java -jar spring-cloud-zookeeper-sample/target/spring-cloud-zookeeper-sample-1.0.0.BUILD-SNAPSHOT.jar --server.port=8081
java -jar spring-cloud-zookeeper-sample/target/spring-cloud-zookeeper-sample-1.0.0.BUILD-SNAPSHOT.jar --server.port=8082
```

示例工程默认使用的端口为 8080。此时的拓扑示意如图 7-2 所示。

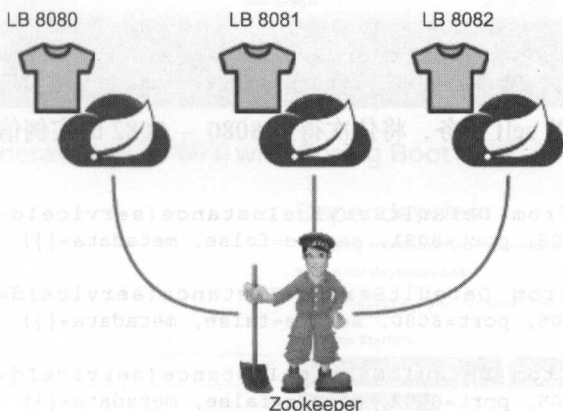


图 7-2 Zookeeper 支持服务高可用示例

该示例的根路径演示了使用 Ribbon 实现负载均衡。不断请求某一实例的根路径，将会依次得到 8080 ~ 8082 的实例信息。如下示例是使用 httpie 对 8080 端口上服务的一次请求。

```
http :8080
{
  "host": "192.168.3.105",
  "metadata": {},
  "port": 8080,
  "secure": false,
  "server": {
    "alive": true,
    "host": "192.168.3.105",
    "hostPort": "192.168.3.105:8080",
    "id": "192.168.3.105:8080",
    "metaInfo": {
      "appName": "testZookeeperApp",
      "instanceId": "1e428cc6-28a8-4597-ace1-85c699d3ae04",
      "serverGroup": null,
      "serviceIdForDiscovery": null
    },
    "port": 8080,
    "readyToServe": true,
    "zone": "UNKNOWN"
  },
  "serviceId": "testZookeeperApp",
  "uri": "http://192.168.3.105:8080"
}
```

从上述返回的结果中，可以看到服务主机（host）、服务端口（port）等信息，这些信息是服务启动时，Spring Cloud Zookeeper 自动注册到 Zookeeper 上的。

该示例提供了 Spring MVC 实现的 RESTful 服务。其中 self 服务演示了使用 Feign 对不同实例的 hi 服务的轮询请求。

多次请求 8080 上的 hi 服务，返回结果都是 8080 实例信息。

```
http :8080/hi
Hello World! from DefaultServiceInstance(serviceId=testZookeeperApp,
host=192.168.3.105, port=8080, secure=false, metadata={})
```

多次请求 8080 上的 self 服务，将依次得到 8080 ~ 8082 的实例信息。

```
http :8080/self
Hello World! from DefaultServiceInstance(serviceId=testZookeeperApp,
host=192.168.3.105, port=8081, secure=false, metadata={})
http :8080/self
Hello World! from DefaultServiceInstance(serviceId=testZookeeperApp,
host=192.168.3.105, port=8080, secure=false, metadata={})
http :8080/self
Hello World! from DefaultServiceInstance(serviceId=testZookeeperApp,
host=192.168.3.105, port=8082, secure=false, metadata={})
```

通过这个简单的示例，我们了解了如何自行开发基于 Spring Cloud Zookeeper 的 RESTful 服务工程，结合 7.2 节的 Bootiful 思想，我们快速构建一个自己的使用 Zookeeper 作为服务发现组件的 RESTful 服务。

3. 快速构建

阅读指南

7.3.1 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/7.3.1.boot.zookeeper>。

使用浏览器打开 <http://start.spring.io>，在 Artifact 输入框中输入 boot.zookeeper 作为项目名称，如图 7-3 所示。我们在右边的输入框中输入 zookeeper 和 jersey，将会得到提示，最终我们选定如下 3 个依赖。

☐ Zookeeper Configuration

☐ Zookeeper Discovery

☐ Jersey

下载并解压快速构建的 boot.zookeeper 项目，增加如下 3 个类。

入口类 BootZookeeperApplication，定义了 Spring Boot 应用配置注解 @SpringBootApplication 和服务发现注解 @EnableDiscoveryClient。


```

@SpringBootApplication
@EnableDiscoveryClient
public class BootZookeeperApplication {
    public static void main(String[] args) {
        SpringApplication.run(BootZookeeperApplication.class, args);
    }
}

```

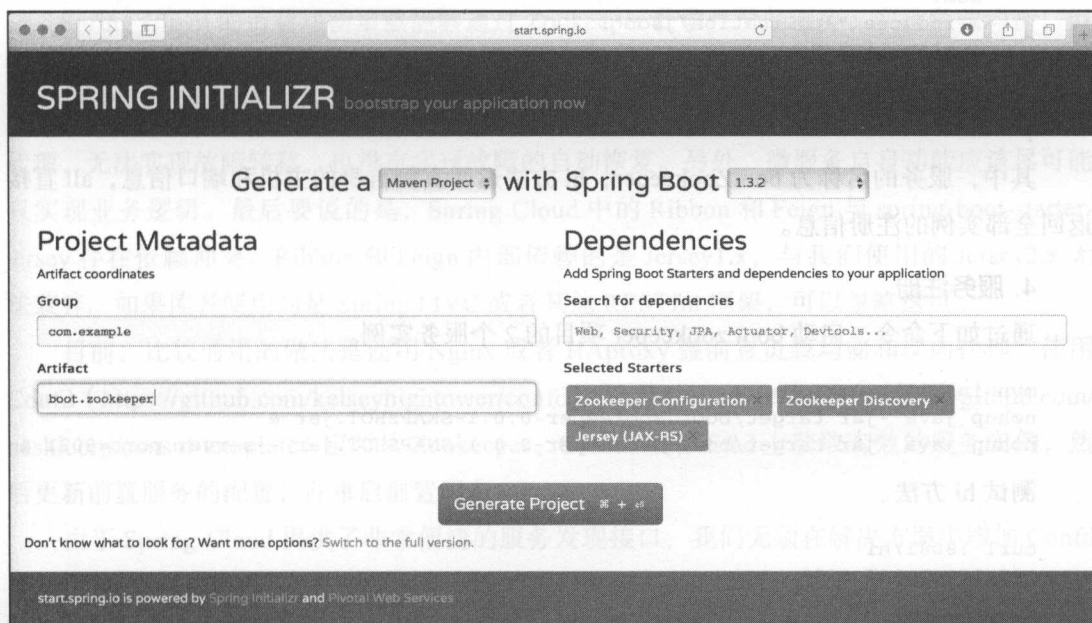


图 7-3 快速创建依赖 Zookeeper 的 Spring Cloud 项目

Jersey 资源配置类 JerseyConfig，注册了 RESTful 服务资源类 MyResource。

```

@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        register(MyResource.class);
    }
}

```

资源类 MyResource 用于定义资源方法，示例如下。

```

@Component
@Path("/")
public class MyResource {
    @Autowired
    private DiscoveryClient discovery;

    @Value("${spring.application.name:bootZookeeper}")
    private String appName;

    @Path("hi")
}

```

```

@GET
public String hi() {
    ServiceInstance serviceInstance = discovery.getLocalServiceInstance();
    return serviceInstance.getHost() + ":" + serviceInstance.getPort();
}

@Path("all")
@GET
@Produces("application/json")
public List<ServiceInstance> getServices() {
    return discovery.getInstances(appName);
}
}

```

其中，服务的名称为 `bootZookeeper`，`hi` 方法返回实例自身的主机和端口信息，`all` 直接返回全部实例的注册信息。

4. 服务注册

通过如下命令，启动 `boot.zookeeper` 项目的 2 个服务实例。

```

mvn install -DskipTests
nohup java -jar target/boot.zookeeper-0.0.1-SNAPSHOT.jar &
nohup java -jar target/boot.zookeeper-0.0.1-SNAPSHOT.jar --server.port=8081 &

```

测试 `hi` 方法。

```

curl :8081/hi
192.168.3.107:8081

```

测试 `all` 方法。

```

curl :8080/all
[{"serviceId":"bootZookeeper","host":"192.168.3.107","port":8081,"secure":false,"metadata":{},"uri":"http://192.168.3.107:8081"},
{"serviceId":"bootZookeeper","host":"192.168.3.107","port":8080,"secure":false,"metadata":{},"uri":"http://192.168.3.107:8080"}]

```

我们可以看到，8080 和 8081 端口上的服务，已经被注册到 `zookeeper` 并被自动发现。通过 `zkCli` 可以查看 `zookeeper` 中保存的注册信息。

```

ls /services/bootZookeeper
[a78fbfc7-6e51-4fd0-9c74-29edb479b283, 257bc4f9-fae0-4719-9dd3-31a95bcb9c02]

```

上面的返回值是 `bootZookeeper` 服务的两个实例 ID。访问其中一个路径，可以获取该实例的详情。

```

get /services/bootZookeeper/a78fbfc7-6e51-4fd0-9c74-29edb479b283
{"name":"bootZookeeper","id":"a78fbfc7-6e51-4fd0-9c74-29edb479b283","address":"192.168.3.107","port":8081,"sslPort":null,"payload":{"@class":"org.springframework.cloud.zookeeper.discovery.ZookeeperInstance","id":"bootZookeeper"}

```

```
er:8081", "name": "bootZookeeper"}, {"registrationTimeUTC": 1455114161021, "serviceType": "DYNAMIC", "uriSpec": {"parts": [{"value": "scheme", "variable": true}, {"value": "://", "variable": false}, {"value": "address", "variable": true}, {"value": ":", "variable": false}, {"value": "port", "variable": true}]}}
```

5. 服务发现

Spring Cloud 的负载均衡虽然能够通过 Zookeeper 获取注册的服务，实现负载均衡，但负载本身存在单点问题。虽然通过访问某个实例（比如端口为 8080 的实例），可以轮询到其他端口（8080 ~ 8082）上的服务，但 8080 自身的单点故障无法保证。因为没有前置的反向代理，无法实现故障转移，也没有实现故障的自动恢复。另外，微服务自身功能应该尽可能只实现业务逻辑。最后要说的是，Spring Cloud 中的 Ribbon 和 Feign 与 spring-boot-starter-jersey 存在依赖冲突。Ribbon 和 Feign 内部依赖的是 Jersey1.x，与我们使用的 Jersey2.x 无法兼容。如果读者使用的是 Spring MVC 或者其他 RESTful 框架，可以忽略这点。

目前，比较通用的做法是使用 Nginx 或者 HAProxy 做前置负载均衡和反向代理，使用 Confd（<https://github.com/kelseyhightower/confd>）或者 Consul Template（<https://github.com/hashicorp/consul-template>）定时从 Zookeeper、Consul 或者 Etcd 上获取有效的服务实例，然后更新前置服务的配置，并重启前置服务。

由于 Spring Cloud 提供了非常便捷的服务发现接口，我们无须在解决方案中增加 Confd 或类似组件，只需利用服务发现接口，根据前置服务（这里以 Nginx 为例）的配置文件格式返回服务信息列表，然后定时更新前置服务就可以了。服务的示例代码如下。

```
@Path("hi2")
@GET
public String hi2() {
    List<ServiceInstance> serviceInstances = getServices();
    StringBuilder result = new StringBuilder(" ");
    for (ServiceInstance s : serviceInstances) {
        result.append("server ").append(s.getHost()).append(":").append(s.getPort()).append(";");
    }
    return result.toString();
}
```

在这段代码中，ServiceInstance 是 Spring Cloud Commons 中的服务实例接口，根据 Nginx 的格式，我们从中获取主机和端口即可。

这个 hi2 服务无论部署在哪个实例上，都依然存在单点问题，也就是不可靠的。因此，在调用 hi2 时须考虑服务失效问题，一个简单的办法是从指定的服务地址或者端口列表的第一个开始，遍历检测该实例的 hi2 服务返回的 HTTP 状态码是否是 200（OK），只有为 200 后，才真正获取该服务的内容，示例代码如下。

```

port=8080
for (( ; ${port} <= 8100; port++ ))
do
    status=$(curl -s -w %{http_code} -o /dev/null :${port}/hi2)
    echo "status=${status}"
    if [ ${status} == 200 ]; then
        break
    fi
done

```

上述代码从 8080 端口开始遍历 hi2 服务的有效性，直至找到有效的实例停止遍历。然后获取服务列表信息，该信息会用来替换接下来的 Nginx 模板中的反向代理信息。

6. Nginx 服务

开发环境下，Nginx 的极简安装和启动示例如下。

```

brew install nginx
==> Installing dependencies for nginx: pcre, openssl

==> Installing nginx dependency: pcre
==> Downloading https://homebrew.bintray.com/bottles/pcre-8.38.el_capitan.
bottle.tar.gz
.....
/usr/local/Cellar/pcre/8.38: 146 files, 5.4M

==> Installing nginx dependency: openssl
==> Downloading https://homebrew.bintray.com/bottles/openssl-1.0.2f.el_
capitan.bottle.tar.gz
.....
/usr/local/Cellar/openssl/1.0.2f: 466 files, 11.9M

==> Installing nginx
==> Downloading https://homebrew.bintray.com/bottles/nginx-1.8.1.el_capitan.
bottle.tar.gz
.....
Docroot is: /usr/local/var/www

The default port has been set in /usr/local/etc/nginx/nginx.conf to 8080 so that
nginx can run without sudo.

nginx will load all files in /usr/local/etc/nginx/servers/.

To have launchd start nginx at login:
  ln -sfv /usr/local/opt/nginx/*.plist ~/Library/LaunchAgents
Then to load nginx now:
  launchctl load ~/Library/LaunchAgents/homebrew.mxcl.nginx.plist
Or, if you don't want/need launchctl, you can just run:
  nginx
==> Summary
/usr/local/Cellar/nginx/1.8.1: 7 files, 946.2K

```

7. 负载均衡和反向代理

Nginx 监听配置项 http-server-listen 中定义的端口，向外提供 HTTP 服务。配置项

location-proxy_pass 中定义了反向代理地址，具体的服务列表在配置项 http-upstream 中定义。一个简单的配置模板示例 nginx_template.conf 如下。

```
worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream bootzookeeper {
        MICRO_SERVICES
    }

    server {
        listen 80;
        server_name localhost;
        location / {
            proxy_pass http://bootzookeeper;
            proxy_redirect off;
        }
    }
}
```

在这段代码中，MICRO_SERVICES 是占位符，内容将被替换为 hi2 服务的返回值。bootzookeeper 是 HTTP 服务背后真正的服务名称。

通过前面第 5 小节讲述的脚本，我们可以得到有效的 hi2 服务地址，请求该服务地址可以得到完整的服务列表，替换配置文件并重启 Nginx 服务的示例如下。

```
if [ ${status} == 200 ]; then
    echo "port=${port}"
    # 关注点 1
    cp ${BASE}/nginx_template.conf ${BASE}/nginx_services.conf
    # 关注点 2
    services=$(curl :${port}/hi2)
    echo "services=${services}"
    # 关注点 3
    sed -i -e "s/MICRO_SERVICES/${services}/g" ${BASE}/nginx_services.conf
    # 关注点 4
    /usr/local/bin/nginx -t -c ${BASE}/nginx_services.conf
    /usr/local/bin/nginx -s reload -c ${BASE}/nginx_services.conf
    echo "Done"
else
    echo "No available services"
fi
```

这段代码中，首先从模板文件 nginx_template.conf 复制内容到最终使用的配置文件 nginx_services.conf，见关注点 1；然后请求 hi2 服务并将返回值赋给 services 变量，见关注点 2；接着使用 services 值替换 nginx_services.conf 的占位符 MICRO_SERVICES，见关注点 3；最后重启 Nginx 服务，见关注点 4。这里要说明的是 Nginx 服务重启非常快，期间的

请求处于 pending 状态直至重启完毕。上述脚本详见 porter.sh。

8. 完整测试

1) 使用 zkServer start 命令启动 Zookeeper, 使用 zkCli -server 192.168.3.107:2181 命令校验 Zookeeper 服务已经启动。

2) 编译 boot.zookeeper 工程, 分别在 8080 ~ 8082 端口启动 bootZookeeper 服务实例, 并启动 Nginx 服务。

```
mvn clean install -DskipTests
nohup java -jar ${BASE}/target/boot.zookeeper-0.0.1-SNAPSHOT.jar &
nohup java -jar ${BASE}/target/boot.zookeeper-0.0.1-SNAPSHOT.jar --server.
port=8081 &
nohup java -jar ${BASE}/target/boot.zookeeper-0.0.1-SNAPSHOT.jar --server.
port=8082 &
cp ${BASE}/shell/nginx_services0.conf ${BASE}/shell/nginx_services.conf
nginx -c ${BASE}/shell/nginx_services.conf
```

nginx_services0.conf 是 Nginx 的初始配置, 内容如下。

```
worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream bootzookeeper {
        server 192.168.3.107:8082;server 192.168.3.107:8081;server
192.168.3.107:8080;
    }

    server {
        listen 8000;
        server_name localhost;
        location / {
            proxy_pass http://bootzookeeper;
            proxy_redirect off;
        }
    }
}
```

此时的组件拓扑示意如图 7-4 所示。

3) 查看已注册的 bootZookeeper 服务, 此时存在 3 个实例。

```
zkCli -server 192.168.3.107:2181

ls /services/bootZookeeper
[679a5027-a817-4602-8ec1-06a7e4670623, 809b7ec0-ab66-4cdd-bc3e-1c2dc3be6236,
cebe845e-976c-48b0-bedb-e68169084776]
```

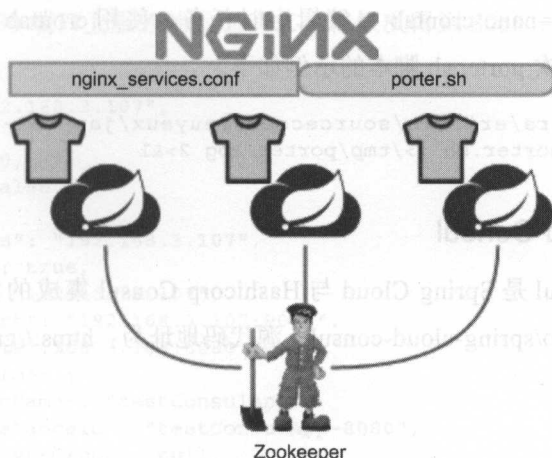


图 7-4 高可用微服务示例之 Zookeeper 版

4) 请求 Nginx 服务，可以看到 Nginx 对服务实例的轮询访问。

```
curl localhost:8000/hi
192.168.3.107:8081
```

```
curl localhost:8000/hi
192.168.3.107:8080
```

```
curl localhost:8000/hi
192.168.3.107:8082
```

5) 停止 8080 端口上的服务，然后执行 `porter.sh`。再次请求 Nginx 服务，可以看到轮询的实例中不再包括 8080 端口的。

```
curl localhost/hi
192.168.3.107:8082
```

```
curl localhost/hi
192.168.3.107:8081
```

6) 查看已注册的 `bootZookeeper` 服务，此时存在两个实例。

```
zkCli -server 192.168.3.107:2181
```

```
ls /services/bootZookeeper
[809b7ec0-ab66-4cdd-bc3e-1c2dc3be6236, cebe845e-976c-48b0-bedb-e68169084776]
```

到此，我们验证了这套高可用方案的可行性。

剩余的工作是将 `porter.sh` 加入 `launchd` 或者 `crontab`，定时更新可用服务列表，重启 Nginx，这样做也能部分解决 Nginx 单点故障的问题。可以使用虚拟 IP 技术构建 Nginx 集群，比如 Keepalive。

可以使用 `EDITOR=nano crontab -e` 编辑定时任务，使用 `crontab -l` 查看当前的定时任务。设定每分钟执行一次 `porter.sh` 脚本的示例如下。

```
* * * * * /Users/erichan/sourcecode/feuyeux/jax-rs2-guide-II/7.3.boot.
zookeeper/shell/porter.sh >>/tmp/porter.log 2>&1
```

7.3.2 Spring Cloud Consul

Spring Cloud Consul 是 Spring Cloud 与 Hashicorp Consul 集成的实现。该项目的地址为：<http://cloud.spring.io/spring-cloud-consul>，源代码地址为：<https://github.com/spring-cloud/spring-cloud-consul>。

1. Consul 服务

开发环境下，Consul 的极简安装命令是 `brew cask install consul`，启动服务的命令示例为：`consul agent -dev -bind 192.168.3.107`，如果服务器存在多网卡，需要指定具体的 IP，本例中的 192.168.3.107 是 `en0`。可以通过访问 `http://localhost:8500` 验证服务是否启动。

2. 官方示例

Spring Cloud Consul 的源代码项目中提供了示例项目 `spring-cloud-consul-sample`。按照 README 的指导，我们启动如下 3 个服务实例。

```
nohup java -jar spring-cloud-consul-sample/target/spring-cloud-consul-sample-
1.0.0.BUILD-SNAPSHOT.jar &
nohup java -jar spring-cloud-consul-sample/target/spring-cloud-consul-sample-
1.0.0.BUILD-SNAPSHOT.jar --server.port=8081 &
nohup java -jar spring-cloud-consul-sample/target/spring-cloud-consul-sample-
1.0.0.BUILD-SNAPSHOT.jar --server.port=8082 &
```

示例工程默认使用的端口为 8080。此时的拓扑示意如图 7-5 所示。

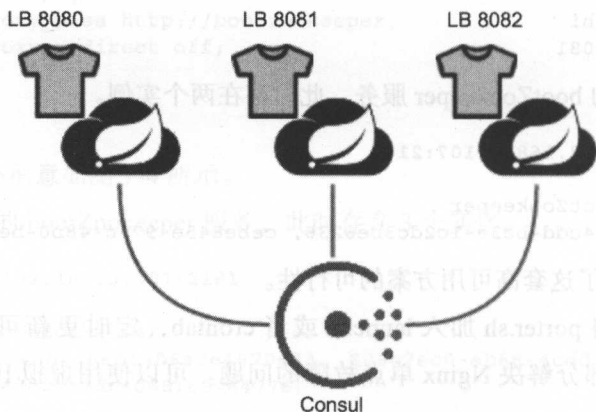


图 7-5 Consul 支持服务高可用示例

使用 httpie 对 8080 端口上服务的发起请求, 将依次得到 8080 ~ 8082 的实例信息。

```
{
  "host": "192.168.3.107",
  "metadata": {},
  "port": 8080,
  "secure": false,
  "server": {
    "address": "192.168.3.107",
    "alive": true,
    "host": "192.168.3.107",
    "hostPort": "192.168.3.107:8080",
    "id": "192.168.3.107:8080",
    "metaInfo": {
      "appName": "testConsulApp",
      "instanceId": "testConsulApp-8080",
      "serverGroup": null,
      "serviceIdForDiscovery": null
    },
    "node": "MacBookPro-4C16G.local",
    "port": 8080,
    "readyToServe": true,
    "zone": "UNKNOWN"
  },
  "serviceId": "testConsulApp",
  "uri": "http://192.168.3.107:8080"
}
```

该示例提供了 Spring MVC 实现的 RESTful 服务。其中, feign 服务演示了使用 Feign 对不同实例的 choose 服务的轮询请求, me 服务提供实例自身信息。

多次请求 8080 上的 me 服务, 返回结果都是 8080 实例信息。

http :8080/me

```
{
  "host": "MacBookPro-4C16G.local",
  "metadata": {},
  "port": 8080,
  "secure": false,
  "serviceId": "testConsulApp-8080",
  "uri": "http://MacBookPro-4C16G.local:8080"
}
```

多次请求 8080 上的 self 服务, 将依次得到 8080 ~ 8082 的实例信息。

http:8080/feign
http://192.168.3.107:8080

http :8080/feign
http://192.168.3.107:8081

http :8080/feign
http://192.168.3.107:8082

3. 快速构建

阅读指南

7.3.2 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/>

7.3.2.boot.consul。

使用浏览器打开 <http://start.spring.io>，在 Artifact 输入框中输入 boot.consul 作为项目名称，如图 7-6 所示。我们在右边的输入框中输入 consul 和 jersey，将会得到提示，最终我们选定如下 3 个依赖。

☐ Consul Configuration

☐ Consul Discovery

☐ Jersey

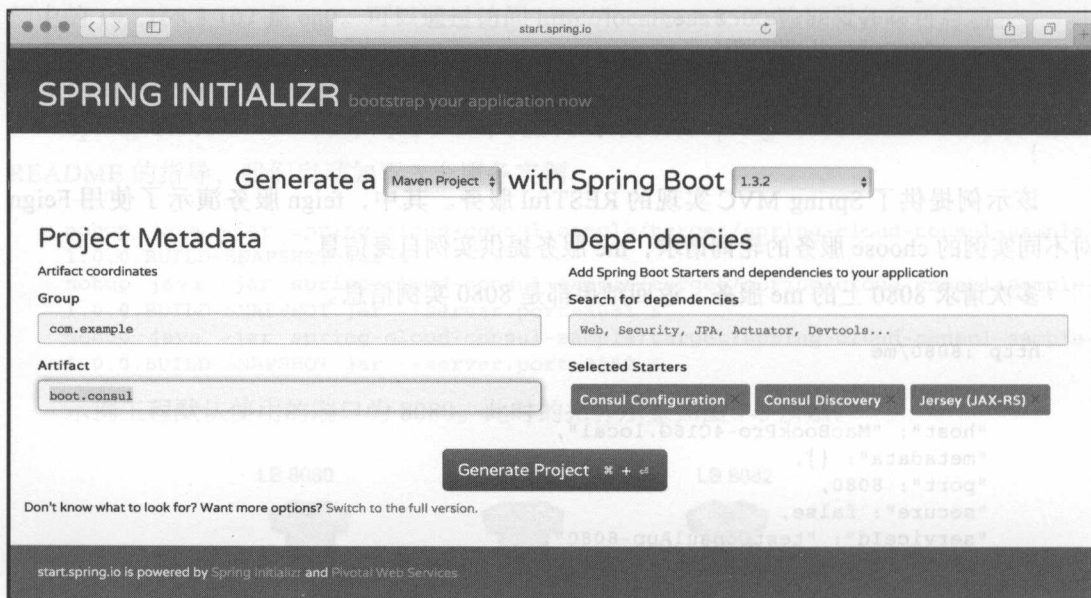


图 7-6 快速创建依赖 Consul 的 Spring Cloud 项目

下载并解压快速构建的 boot.consul 项目，增加 BootConsulApplication、JerseyConfig、MyResource 这 3 个类，其中，服务的名称为 bootConsul，hi 方法返回实例自身的主机和端口信息，all 直接返回全部实例的注册信息。请参考 7.3.1 节的第 3 小节。

4. 服务注册

通过如下命令，启动 boot.consul 项目的 2 个服务实例。


```
mvn install -DskipTests
nohup java -jar target/boot.consul-0.0.1-SNAPSHOT.jar &
nohup java -jar target/boot.consul-0.0.1-SNAPSHOT.jar --server.port=8081 &
```

测试 hi 方法。

```
curl :8081/hi
MacBookPro-4C16G.local:8081
```

测试 all 方法。

```
curl :8080/all
[{"serviceId":"bootConsul","host":"192.168.3.107","port":8080,"secure":false,"
metadata":{},"uri":"http://192.168.3.107:8080"},
{"serviceId":"bootConsul","host":"192.168.3.107","port":8081,"secure":false,"
metadata":{},"uri":"http://192.168.3.107:8081"}]
```

我们可以看到，8080 和 8081 端口上的服务，已经被注册到 consul 并被自动发现。通过访问 <http://localhost:8500/v1/catalog/service/bootConsul>，可以查看 consul 中保存的注册信息。

```
[{"Node":"MacBookPro-4C16G.local","Address":"192.168.3.107","ServiceID":"bootC
onsul-8080","ServiceName":"bootConsul","ServiceTags":[],"ServiceAddress":"192.
168.3.107","ServicePort":8080,"ServiceEnableTagOverride":false,"CreateIndex":4
4,"ModifyIndex":44},
{"Node":"MacBookPro-4C16G.local","Address":"192.168.3.107","ServiceID":"bootC
onsul-8081","ServiceName":"bootConsul","ServiceTags":[],"ServiceAddress":"192.
168.3.107","ServicePort":8081,"ServiceEnableTagOverride":false,"CreateIndex":4
5,"ModifyIndex":45}]
```

更多内容请参考 <https://www.consul.io/docs/agent/http/catalog.html>。

5. 服务发现

请参考 7.3.1 节的第 5 小节。hi2 服务测试结果如下。

```
curl :8080/hi2
server 192.168.3.107:8080;server 192.168.3.107:8081;
```

6. 完整测试

1) 使用 `consul agent -dev -bind 192.168.3.107` 命令启动 Consul，通过访问 <http://localhost:8500> 校验服务已经启动。

2) 编译 boot.consul 工程，分别在 8080-8082 端口启动 bootConsul 服务实例，并启动 Nginx 服务。

```
mvn clean install -DskipTests
nohup java -jar ${BASE}/target/boot.consul-0.0.1-SNAPSHOT.jar &
nohup java -jar ${BASE}/target/boot.consul-0.0.1-SNAPSHOT.jar --server.
port=8081 &
nohup java -jar ${BASE}/target/boot.consul-0.0.1-SNAPSHOT.jar --server.
port=8082 &
```

```

cp ${BASE}/shell/nginx_services0.conf ${BASE}/shell/nginx_services.conf
nginx -c ${BASE}/shell/nginx_services.conf

nginx_services0.conf 是 Nginx 的初始配置，内容如下。

worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream bootconsul {
        server 192.168.3.107:8082;server 192.168.3.107:8081;server
192.168.3.107:8080;
    }

    server {
        listen 80;
        server_name localhost;
        location / {
            proxy_pass http://bootconsul;
            proxy_redirect off;
        }
    }
}

```

此时的组件拓扑示意图如图 7-7 所示。

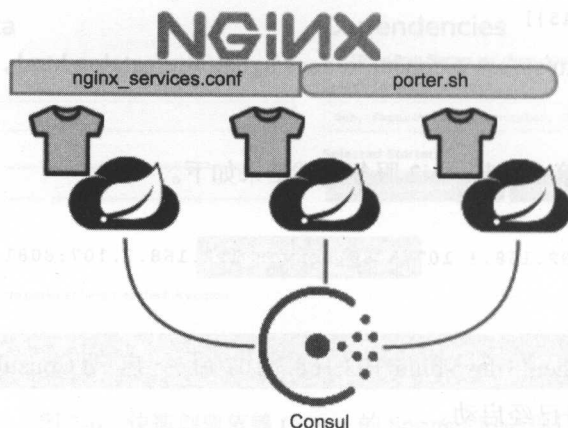


图 7-7 高可用微服务示例之 Consul 版

3) 启动定时器触发 porter 脚本。

```

EDITOR=nano crontab -e
* * * * * /Users/erichan/sourcecode/feuyeux/jax-rs2-guide-II/7.3.boot.consul/
shell/porter.sh >>/tmp/porter.consul.log 2>&1 \ \ \

```

4) 停止 8080 端口上的服务，启动 8083-8085 端口上的服务。通过 curl :8500/v1/

catalog/service/bootConsul 命令观察 Consul 已注册服务。

```
{
  "Node": "MacBookPro-4C16G.local", "Address": "192.168.3.107", "ServiceID": "bootConsul-8081", "ServiceName": "bootConsul", "ServiceTags": [], "ServiceAddress": "192.168.3.107", "ServicePort": 8081, "ServiceEnableTagOverride": false, "CreateIndex": 11, "ModifyIndex": 11},
  {
    "Node": "MacBookPro-4C16G.local", "Address": "192.168.3.107", "ServiceID": "bootConsul-8082", "ServiceName": "bootConsul", "ServiceTags": [], "ServiceAddress": "192.168.3.107", "ServicePort": 8082, "ServiceEnableTagOverride": false, "CreateIndex": 12, "ModifyIndex": 12},
    {
      "Node": "MacBookPro-4C16G.local", "Address": "192.168.3.107", "ServiceID": "bootConsul-8083", "ServiceName": "bootConsul", "ServiceTags": [], "ServiceAddress": "192.168.3.107", "ServicePort": 8083, "ServiceEnableTagOverride": false, "CreateIndex": 22, "ModifyIndex": 22},
      {
        "Node": "MacBookPro-4C16G.local", "Address": "192.168.3.107", "ServiceID": "bootConsul-8084", "ServiceName": "bootConsul", "ServiceTags": [], "ServiceAddress": "192.168.3.107", "ServicePort": 8084, "ServiceEnableTagOverride": false, "CreateIndex": 24, "ModifyIndex": 24},
        {
          "Node": "MacBookPro-4C16G.local", "Address": "192.168.3.107", "ServiceID": "bootConsul-8085", "ServiceName": "bootConsul", "ServiceTags": [], "ServiceAddress": "192.168.3.107", "ServicePort": 8085, "ServiceEnableTagOverride": false, "CreateIndex": 25, "ModifyIndex": 25}
        ]
      }
```

这里需要注意的是不要使用 `kill -9 $PID`，因为这样会让服务进程没有机会去 Consul 注销自己。启动、测试和停止一个服务的示例，请参考如下命令。

```
nohup java -jar ${BASE}/target/boot.consul-0.0.1-SNAPSHOT.jar &
echo $! > boot.consul.8080.pid
sleep 10
curl :8080/hi
sleep 2
curl :8080/hi2
sleep 2
kill -15 `cat boot.consul.8080.pid`
```

5) 请求 hi2 服务，查看当前可用服务列表。

```
curl :8080/hi2
server 192.168.3.107:8081;server 192.168.3.107:8082;server
192.168.3.107:8083;server 192.168.3.107:8084;server 192.168.3.107:8085;
```

7.3.3 Spring Cloud Etcd

Spring Cloud Etcd 是 Spring Cloud 与 CoreOS Etcd 集成的实现。该项目目前处于孵化阶段，尚未进入 Spring Cloud 的 Angel 版本（对应 Spring Boot 的 1.2.x）和 Brixton 版本（对应 Spring Boot 的 1.3.）。源代码地址为：<https://github.com/spring-cloud-incubator/spring-cloud-etcd>。

1. Etcd 服务

开发环境下，Etcd 的极简安装示例如下。

```
brew install etcd
```

```
==> Downloading https://homebrew.bintray.com/bottles/etcd-2.2.5.el_capitan.
      bottle.tar.gz
```

```
.....
```

```
To have launchd start etcd at login:
```

```
ln -sfv /usr/local/opt/etcd/*.plist ~/Library/LaunchAgents
```

```
Then to load etcd now:
```

```
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.etcd.plist
```

```
==> Summary
```

```
/usr/local/Cellar/etcd/2.2.5: 7 files, 42.3M
```

使用 etcd 启动 Etcd 服务，通过访问 <http://localhost:2379/version> 快速验证服务是否启动。

```
{"etcdserver":"2.2.5","etcdcluster":"2.2.0"}
```

2379 端口是 Etcd2 开始启用的标准端口，用来代替此前的 4001 端口。

2. 官方示例

Spring Cloud Etcd 的源代码项目中提供了示例项目 `spring-cloud-etcd-sample`。按照 README 的指导，我们启动如下 3 个服务实例。

```
./mvnw --settings .settings.xml install -DskipTests=true
```

```
nohup java -jar spring-cloud-etcd-sample/target/spring-cloud-etcd-sample-
1.0.0.BUILD-SNAPSHOT.jar &
```

```
nohup java -jar spring-cloud-etcd-sample/target/spring-cloud-etcd-sample-
1.0.0.BUILD-SNAPSHOT.jar --server.port=8081 &
```

```
nohup java -jar spring-cloud-etcd-sample/target/spring-cloud-etcd-sample-
1.0.0.BUILD-SNAPSHOT.jar --server.port=8082 &
```

示例工程默认使用的端口为 8080。此时的拓扑示意图如图 7-8 所示。

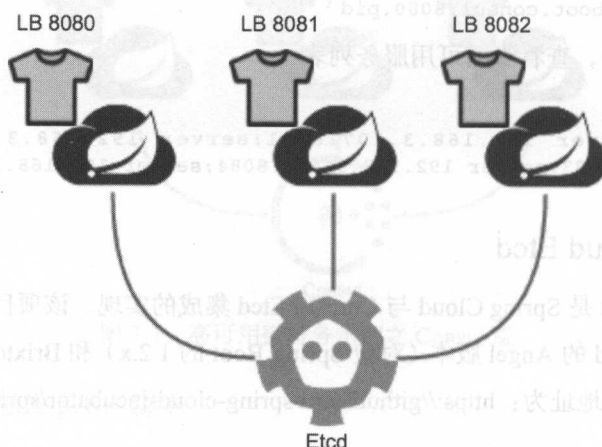


图 7-8 Etcd 支持服务高可用示例

使用 `httpie` 对 8080 端口上服务的发起请求，将依次得到 8080 ~ 8082 的实例信息。

```

{
  "serviceId": "testEtcdApp",
  "server": {
    "host": "192.168.100.40",
    "port": 8080,
    "id": "192.168.100.40:8080",
    "zone": "UNKNOWN",
    "readyToServe": true,
    "metaInfo": {
      "appName": "testEtcdApp",
      "serviceIdForDiscovery": null,
      "instanceId": "testEtcdApp:8080",
      "serverGroup": null
    },
    "alive": true,
    "hostPort": "192.168.100.40:8080"
  },
  "secure": false,
  "metadata": {},
  "host": "192.168.100.40",
  "port": 8080,
  "uri": "http://192.168.100.40:8080"
}

```

该示例提供了 Spring MVC 实现的 RESTful 服务。其中 all 服务提供了服务实例列表，me 服务提供实例自身信息。

使用 http:8080/me 请求 8080 上的 me 服务，返回 8080 实例信息。

```

{
  "host": "192.168.100.40",
  "metadata": {},
  "port": 8080,
  "secure": false,
  "serviceId": "testEtcdApp",
  "uri": "http://192.168.100.40:8080"
}

```

使用 http:8080/all 请求 8080 上的 all 服务，返回服务实例列表信息。

```

[
  {
    "host": "192.168.100.40",
    "metadata": {},
    "port": 8080,
    "secure": false,
    "serviceId": "testEtcdApp",
    "uri": "http://192.168.100.40:8080"
  },
  {
    "host": "192.168.100.40",
    "metadata": {},
    "port": 8081,
    "secure": false,
    "serviceId": "testEtcdApp",

```



```

        "uri": "http://192.168.100.40:8081"
    },
    {
        "host": "192.168.100.40",
        "metadata": {},
        "port": 8082,
        "secure": false,
        "serviceId": "testEtcdApp",
        "uri": "http://192.168.100.40:8082"
    }
]

```

3. 快速构建

阅读指南

7.3.3 节的示例源代码地址为: <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/>

7.3.3.boot.etcd。

与 7.3.1 节一样, 我们使用 3 个依赖。

❑ Etcd Configuration

❑ Etcd Discovery

❑ Jersey

由于本书截稿时 Spring Cloud Etcd 处于孵化状态, 因此无法通过 <http://start.spring.io> 创建(读者阅读到此时, 可以先搜索下 Spring Cloud Etcd 是否已经成为正式项目。既无奈又兴奋的是, 我们搞技术的必须要拥抱日新月异的技术变化), 手动定义服务工程 boot.etcd 的依赖也非常容易。

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-etcd-config</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-etcd-discovery</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
    <exclusions>
        <exclusion>
            <groupId>com.netflix.ribbon</groupId>
            <artifactId>ribbon-httpclient</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>

```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jersey</artifactId>
<version>1.3.2.RELEASE</version>
</dependency>

```

增加 BootEtcdApplication、JerseyConfig、MyResource 这 3 个类，其中，服务的名称为 bootEtcd，hi 方法返回实例自身的主机和端口信息，all 直接返回全部实例的注册信息。请参考 7.3.1 节的第 3 小节。

4. 服务注册

设置 server.port: 0 随机分配端口，然后通过如下命令，启动 boot.etcd 项目的两个服务实例。

```

mvn install -DskipTests
export SERVER_PORT=8081 && nohup java -jar target/boot.etcd-0.0.1-SNAPSHOT.jar &
echo $! > boot.etcd.1.pid
export SERVER_PORT=8082 && nohup java -jar target/boot.etcd-0.0.1-SNAPSHOT.jar &
echo $! > boot.etcd.2.pid
lsof -i -n -P | grep 'cat boot.etcd.1.pid'
java 7443 erichan 85u IPv6 0xb38b7a2c363ba795 0t0 TCP *:8081 (LISTEN)

```

测试 hi 方法。

```

curl :8081/hi
192.168.100.12:8081

```

测试 all 方法。

```

curl :8081/all
[{"serviceId":"bootEtcd","host":"192.168.100.12","port":8081,"secure":false,"metaData":{},"uri":"http://192.168.100.12:8081"},
{"serviceId":"bootEtcd","host":"192.168.100.12","port":8082,"secure":false,"metaData":{},"uri":"http://192.168.100.12:8082"}]

```

我们可以看到，8081 和 8082 端口上的服务，已经被注册到 etcd 并被自动发现。通过访问 <http://localhost:2379/v2/keys/spring/cloud/discovery/bootEtcd>，可以查看 consul 中保存的注册信息。

```

{"action":"get","node":{"key":"/spring/cloud/discovery/bootEtcd","dir":true,"nodes":[{"key":"/spring/cloud/discovery/bootEtcd/bootEtcd:8081","value":"192.168.100.12:8081","expiration":"2016-02-13T12:55:12.910851698Z","ttl":7,"modifiedIndex":161,"createdIndex":161},{key":"/spring/cloud/discovery/bootEtcd/bootEtcd:8082","value":"192.168.100.12:8082","expiration":"2016-02-13T12:55:12.921770746Z","ttl":7,"modifiedIndex":162,"createdIndex":162}],"modifiedIndex":4,"createdIndex":4}}

```

更多内容请参考 <https://coreos.com/etcd/docs/latest/api.html#listing-a-directory>。

5. 服务发现

请参考 7.3.1 节的第 5 小节。hi2 服务测试结果如下。

```
curl :8081/hi2
server 192.168.100.12:8081;server 192.168.100.12:8082;
```

6. 完整测试

1) 使用 etcd 命令启动 Consul, 通过访问 <http://localhost:2379/version> 校验服务已经启动。

2) 编译 boot.etcd 工程, 分别在 8080-8082 端口启动 bootEtcd 服务实例, 并启动 NGINX 服务。

```
mvn clean install -DskipTests
export SERVER_PORT=8080 && nohup java -jar ${BASE}/target/boot.etcd-0.0.1-SNAPSHOT.jar &
export SERVER_PORT=8081 && nohup java -jar ${BASE}/target/boot.etcd-0.0.1-SNAPSHOT.jar &
export SERVER_PORT=8082 && nohup java -jar ${BASE}/target/boot.etcd-0.0.1-SNAPSHOT.jar &
cp ${BASE}/shell/nginx_services0.conf ${BASE}/shell/nginx_services.conf
nginx -c ${BASE}/shell/nginx_services.conf
```

nginx_services0.conf 是 NGINX 的初始配置, 内容如下。

```
worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream bootzookeeper {
        server 192.168.3.107:8082;server 192.168.3.107:8081;server 192.168.3.107:8080;
    }

    server {
        listen 80;
        server_name localhost;
        location / {
            proxy_pass http://bootzookeeper;
            proxy_redirect off;
        }
    }
}
```

此时的组件拓扑示意图如图 7-9 所示。

3) 启动定时器触发 porter 脚本。

```
EDITOR=nano crontab -e

* * * * * /Users/erichan/sourcecode/feuyeux/jax-rs2-guide-II/7.3.boot.etcd/
shell/porter.sh >>/tmp/porter.etcd.log 2>&1
```

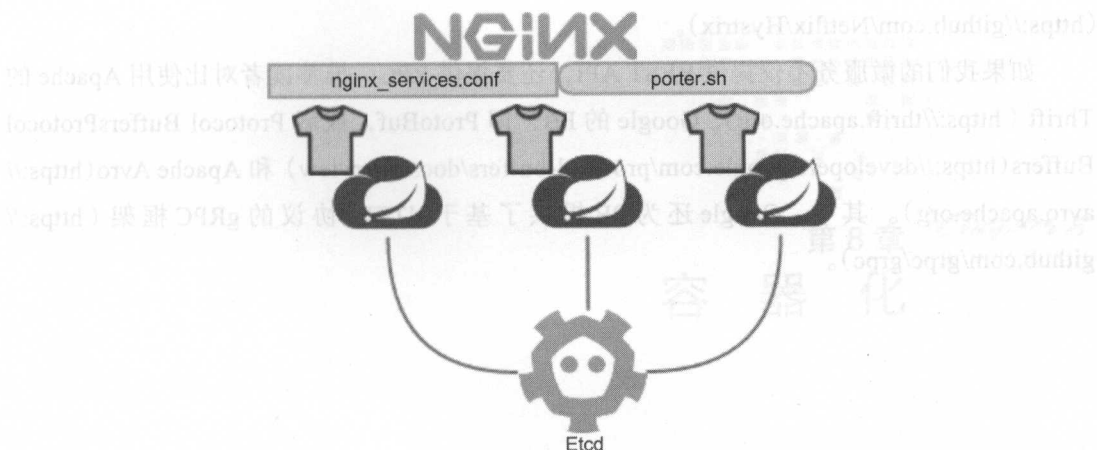


图 7-9 高可用微服务示例之 Etcd 版

4) 停止 8082 端口上的服务，启动 8085 端口上的服务。通过 `curl :2379/v2/keys/spring/cloud/discovery/bootEtcd` 命令观察 Etcd 已注册服务。

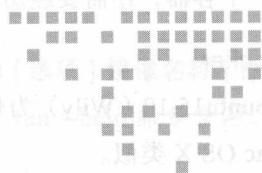
```
.....
[{"key":"/spring/cloud/discovery/bootEtcd/bootEtcd:8081","value":"192.168.100.12:8081","expiration":"2016-02-13T13:10:43.44023393Z","ttl":23,"modifiedIndex":251,"createdIndex":251},
{"key":"/spring/cloud/discovery/bootEtcd/bootEtcd:8080","value":"192.168.100.12:8080","expiration":"2016-02-13T13:10:43.782350024Z","ttl":23,"modifiedIndex":252,"createdIndex":252},
{"key":"/spring/cloud/discovery/bootEtcd/bootEtcd:8085","value":"192.168.100.12:8085","expiration":"2016-02-13T13:10:45.398815826Z","ttl":25,"modifiedIndex":253,"createdIndex":253}]
```

5) 请求 hi2 服务，查看当前可用服务列表。

```
curl :8080/hi2
server 192.168.100.12:8081;server 192.168.100.12:8080;server
192.168.100.12:8085;
```

7.4 本章小结

本章以 REST 服务为出发点，通过简单易用的 Spring Boot 和 Spring Cloud 框架将基本的 REST 服务向微服务方向推进。但微服务风格的内容远不止于此，需要专门的著作来阐述。除了微服务的理论，读者还应该掌握服务底层通信，比如如何使用消息系统实现作业调度、流式计算等，这方面可以结合 Spring 引入基于 AMQP (Advanced Message Queuing Protocol, 高级消息队列协议) 的消息系统。再比如如何使用断路器模式 (Circuit Breaker Pattern) 增强服务单点等健壮性，这方面可以结合 Spring Cloud 引入 Netflix 的 Hystrix



第 8 章 Chapter 8

容器化

如今，以 Docker、CoreOS、Kubernetes 等为代表的容器技术正以迅猛的速度发展着，容器生态圈欣欣向荣。同时，容器化是 REST 服务新的部署形态，有了容器技术，REST 服务焕发了新的生命力。本章将以 REST 以及微服务的部署为目标，交代必要的 Docker 知识，并着重讲述微服务的容器化实战。本书在 Docker 以及容器技术上的讲述非常有限，若读者对这个领域有兴趣，还请阅读专门的书籍和资料。

8.1 容器技术

容器化实践之路是通过对容器技术的认知、理解，并将其用于服务的部署来实现的。本节将从容器、Docker 组件和容器文化 3 个角度概述容器技术，为容器化微服务打好基础。

8.1.1 容器

什么是容器？狭义地说，就是 Docker 镜像的运行时，是宿主机操作系统上的一个进程。从广义上说，虽然 Docker 是当今容器技术的事实标准，但容器技术不限于 Docker 一家。CoreOS 曾经提出 appc (Application Container Spec, 应用容器规范)，并使用 RKT 产品实现，得到了包括谷歌、VMware 和 Red Hat 在内的厂商的支持。一时间，引发了容器领域的轩然大波。随后，各路利益相关的厂商和解，并成立 Open Container Initiative (OCI) 组织规范容器的发展，Docker 为此贡献了 runC 规范。

容器的用途非常广泛，学习成本低，启动一个容器通常只需要简单的一条命令。但是，

从 0 到 1 启动一个容器，还需要经历以下步骤。

1. 安装 Docker

这里以 Ubuntu15.10 (Wily) 为例，其他 Linux 发行版类似，Mac OS X 参见 8.1.2 节，Windows 与 Mac OS X 类似。

如下代码段参考 Docker 官方文档：<https://docs.docker.com/engine/installation/linux/ubuntu/linux/>。

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
58118E89F3A912897C070ADB76221572C52609D
sudo nano /etc/apt/sources.list.d/docker.list
deb https://apt.dockerproject.org/repo ubuntu-wily main
sudo apt-get update
sudo apt-get install linux-image-extra-$(uname -r)
sudo apt-get install docker-engine
sudo service docker start
```

以上命令最终会安装 Docker Engine 并启动 Docker 服务。使用 `sudo docker version` 命令校验 Docker 服务和 Docker 客户端命令是否生效，结果示例如下。

```
Client:
Version:      1.10.1
API version:  1.22
Go version:   go1.5.3
Git commit:   9e83765
Built:        Thu Feb 11 19:32:54 2016
OS/Arch:      linux/amd64
```

```
Server:
Version:      1.10.1
API version:  1.22
Go version:   go1.5.3
Git commit:   9e83765
Built:        Thu Feb 11 19:32:54 2016
OS/Arch:      linux/amd64
```

2. 下载镜像

下载镜像的命令是 `sudo docker pull 镜像名称`。你也可以省略这一步，直接运行第 3 步，但实际执行过程中，这一步还是要执行的。Docker 会检查本地是否存在我们要运行的容器所使用的镜像，如果不存在就从远程仓库拉取到本地。同理，如果我们执行这条命令，而相应的镜像已经存在本地，Docker 将直接使用本地镜像。

镜像名称的格式通常是命名空间 / 镜像用途名称 : 镜像版本。查看本地镜像列表的命令是 `sudo docker images`。

3. 运行容器

运行容器的命令相对丰富，基本格式是 `docker run [选项] 镜像名称 [命令] [参数 ...]`，本章后续会有更详细的演示，也可以通过 `sudo docker run --help` 命令学习。启动 `feuyeux/ubuntu` 镜像的容器并查看 Linux 内核版本的示例如下。

```
sudo docker run -ti --rm feuyeux/ubuntu uname -a
Linux 7e954a75db59 4.2.0-27-generic #32-Ubuntu SMP Fri Jan 22 04:49:08 UTC
2016 x86_64 x86_64 x86_64 GNU/Linux
```

8.1.2 Docker 技术栈

所谓安装了 Docker，其实是指安装了 Docker 的 Engine 和客户端命令行。Docker 自身提供了多方面的组件，概述如下。

1. Docker Engine

Docker Engine 是 Docker 的大脑，我们执行命令创建镜像和运行容器，其背后真正起作用的是 Engine，我们输入的 Docker 命令只是个壳。Engine 底层依赖于 Linux 内核，也就是说目前 Engine 只能安装在 Linux 的各种发行版上。Mac OS X 和 Windows 上无法直接运行 Engine，需要借助一个装有 Linux 操作系统的虚拟机来伪造。这个虚拟机被称为 Driver，运行其上的是 Docker Machine (boot2docker)。

2. Docker Machine

Docker Machine 以主机的形态封装了装有 Docker Engine 的虚拟机实例。Machine 的出现是为了替代 boot2docker，以统一全操作系统上的 Docker 管理。Machine 的本地底层 Driver 是 Oracle VirtualBox，云提供商 AWS、Microsoft Azure、Google Compute Engine 和 Digital Ocean 等也提供了对 Docker Machine 的支持。相对于 Docker Swarm 而言，Machine 创建的虚拟机实例是 Docker 集群上的一个节点 (Node)。

3. Docker Swarm

Docker Swarm 用于管理集群上的 Node 以及容器调度。Swarm 的出现，渗透着 Docker 一统容器生态圈的野心。对于开发者来说，这一趋势的好处是不用额外找寻解决方案，只要跟进 Docker 自身的发展，就基本够用了；坏处是被绑死在一家厂商身上，如果 Docker 产品线能解决的问题有局限，我们很难跳出来。

4. Docker Compose

Docker Compose 的前身是 fig，用于编排多个容器。其定位是为开发者提供开发和测试环境，以节省时间、提高效率。Compose 不适用于大型生产环境的部署。示例详见 8.3.2 节。

5. Docker Toolbox

Docker Toolbox 是为 Mac OS X 和 Windows 操作系统提供的一个工具箱，里面包含了 Engine、Machine、Compose 和镜像图形管理工具 Kitematic。安装 Docker Toolbox 后，启动菜单会多出 2 个图标，分别是 Docker Quickstart Terminal 和 Kitematic。

Docker Quickstart Terminal 用于启动本地 Docker 服务，单击该图标后，执行动作如下。

```
# 1 启动 Docker Machine
Creating machine...
# 2 拷贝 boot2docker.iso 到名为 default 的虚拟机实例目录下
(default) Copying /Users/han/.docker/machine/cache/boot2docker.iso to /Users/
han/.docker/machine/machines/default/boot2docker.iso...
# 3 创建虚拟机实例
(default) Creating VirtualBox VM...
# 4 SSH 登录设定
(default) Creating SSH key...
.....
# 5 启动完成, default 虚拟主机 192.168.99.100 开始提供 Docker 服务
docker is configured to use the default machine with IP 192.168.99.100
```

此时，通过 `docker info` 命令查看当前 Docker 环境的信息，结果示例如下。

```
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 1.10.2
Storage Driver: aufs
  Root Dir: /mnt/sda1/var/lib/docker/aufs
.....
Execution Driver: native-0.2
Logging Driver: json-file
Plugins:
  Volume: local
  Network: bridge null host
Kernel Version: 4.1.18-boot2docker
.....
Debug mode (server): true
.....
Docker Root Dir: /mnt/sda1/var/lib/docker
Labels:
  provider=virtualbox
```

6. Docker Registry 和 Docker Hub

Docker Registry 提供了镜像管理服务。Docker Hub 是 Registry 服务之上的权限管理系统。用户登录后可以管理自己提交的镜像。与 Maven 私服、Gitlab 一样，这部分事情通常不是开发者关心的，作为开发者，知道如何使用命令拉取和推送镜像即可。

国内开发者直接从 Docker Hub 上下载镜像是件非常耗时的事情，建议使用国内优秀的 Docker 云提供商为我们提供的加速器，并将其加入到 docker 配置文件中。其原理是在

docker 运行时, 增加一个 registry-mirror 选项。配置示例如下。

(1) Ubuntu | Debian

```
echo "DOCKER_OPTS=\"\$DOCKER_OPTS --registry-mirror=http://用户名.Docker云域名\"
\" | sudo tee -a /etc/default/docker
sudo service docker restart
```

(2) Boot2Docker On Windows

启动 Boot2docker Start Shell

```
sudo "sh -c \"echo EXTRA_ARGS='--registry-mirror=http://用户名.Docker云域名\"
>> /var/lib/boot2docker/profile\""
```

重新启动 Boot2Docker

(3) Boot2Docker On Mac

```
boot2docker ssh sudo "sh -c \"echo EXTRA_ARGS='--registry-mirror=http://用户名.Docker云域名\"
>> /var/lib/boot2docker/profile\""
boot2docker restart
```

(4) Toolbox On Mac

```
docker-machine ssh default
sudo sed -i "s|EXTRA_ARGS='|EXTRA_ARGS='--registry-mirror=http://用户名.Docker云域名
|g" /var/lib/boot2docker/profile
exit
docker-machine restart default
```

8.1.3 容器文化

Docker 的发展孕育了容器文化, 人们一提到 Docker, 经常会将其与简单、快乐、高效联系在一起。容器文化这个话题足可以放大到一个章节甚至是一本书, 本节的目的是为了正确地理解和使用容器。一般来讲, 容器文化包含以下几个特点。

1. 极简式

构建镜像的目的是为了复用。因为镜像是分层存储的。举个例子, 如果你需要构建一个 Tomcat 镜像, 可以继承自 Java 镜像, 在此基础上只需要下载、安装和配置 Tomcat 即可, Java 环境、操作系统环境都可以从 Tomcat 这层下面的各个层复用过来。同时, 版本化、标签化的镜像, 还能简化构建问题的追溯。比如 Maven 的官方镜像 (https://hub.docker.com/_/maven) 为 Java 版本 1.8、Maven 版本 3.3.3 的镜像提供了多个标签 (3.3.3-jdk-8、3.3.3、3.3-jdk-8、3.3、3-jdk-8、3、latest), 它们对应的镜像是同一个, 如果使用和复用中出现问题, 我们可以很清楚地知道, 这与 Java1.8 或者 Maven3.3.3 有关。

镜像简化了多套环境的统一配置, 为持续集成带来了一致性的测试环境, 为运维提供了跨操作系统、云提供商系统的统一部署能力。

容器的轻量性，为服务的可伸缩性提供了快速扩容和快速回收的能力。

2. 不变式

配置结束于运行前，数据共享于主机卷。我们不要试图修改容器，因为这样破坏了从 Dockerfile 构建镜像，从镜像启动容器的幂等性。如果我们发现容器中的配置、版本、功能等需要改动，请修改相应的 Dockerfile。容器中，服务的配置和产生的数据，通常使用 `-v` 或者 `VOLUME`，从而让容器有读写主机目录的能力，从主机读取配置，将数据写到主机磁盘上。这样做的好处是一个容器挂死后，我们可以轻松地（使用相同参数的命令）启动一个新的容器来代替前者，数据不会丢失。一句话总结，容器负责计算，主机负责存储。

3. 收敛式

一容器一职责，阅后即焚。单一职责的容器与马丁·福勒在微服务中提出的服务单一化、通过 HTTP RESTful API 或者队列实现通信的机制相当吻合。我们在定义 Dockerfile 时，应遵循这个原则：确保容器行为的单一性。每个单一功能的容器组合在一起，与微服务之间的通信非常类似，哪里是瓶颈，哪里出现问题很容易判断、维护和改进。使用 `--rm` 可以让我们实现一次运行并随即丢弃容器。一句话总结，容器一旦启动，它的职责就明确不变，我们所关心的是这个职责下产生的结果。

4. 可执行的容器

基于以上 3 点，我们可以让容器代替可执行文件，完成同样的功能。这里还以 Maven 为例，比如我们同时要支持多个项目，而每个项目所使用的 Java 和 Maven 版本都不同，那么在开发机上为它们安装每种版本及其组合是件让人崩溃的事。使用 Docker 容器能简化这一切，而且我们本机无需安装那么多版本，示例如下。

```
docker run --rm \
-v $(pwd):/project \
-v ~/.m2:/root/.m2 \
-w /project \
maven:3.3.3-jdk-8 mvn install
```

上述命令做了如下的事情。

- ❑ `docker run` 启动 `maven:3.3.3-jdk-8` 镜像的一个容器。在容器的 CMD 中执行了 `mvn install` 命令。
- ❑ `--rm` 将在执行完毕后删除容器。
- ❑ `-v $(pwd):/project` 将当前目录挂载到容器中，映射为 `/project` 目录。这样一来，容器就可以读写主机系统的当前目录了。
- ❑ `-v ~/.m2:/root/.m2` 将用户仓库与镜像中的仓库映射，构建过程中会将当前项目依赖的

包存储于此，这样一来，在容器被废弃掉后，构建成果还能得以保存。

❑ `-w /project` 设置了 `/project` 作为工作目录。这意味着执行 `mvn` 命令将在 `project` 目录中有效。

8.2 REST 服务与容器

Spring Boot 框架将我们带入快速实现微服务的时代，我们可以快速开发、调试、部署一个微服务，但是我们无法惬意地完成服务治理、水平伸缩这样的工作。此时，我们需要将 Spring Boot 与容器技术相结合，通过编排容器之间的关系完成服务依赖、通过快速启动和销毁容器实现服务注册和服务发现，从而实现伸缩性和可用性。

本节首先示范如何将我们熟悉的 REST 服务以镜像的形式分发，以容器的形式运行，然后通过一个支持缓存的 REST 服务，示范如何使用 Docker Compose 编排开发自测阶段的服务依赖。

8.2.1 开始容器化之路

基于 Spring Boot 的 REST 服务的分发形态是一个 jar 包，存在形态是一个 jvm 进程。容器化的过程就是将这样的 jar 构建到 Docker 镜像中，以 image 的形态分发，以 container 的形态运行。

阅读指南

8.2 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/>

8.2.containerization。

1. REST 服务

在思考容器化之前，我们首先要开发和测试 REST 服务。本节的示例非常简单，入参是 HTTP GET 请求，资源路径为 `hi`，出参是 JSON 格式的数组，包含两个字符串元素，示例如下。

```
@Path("hi")
public class DemoResource {
    @GET
    @Produces("application/json")
    public List<String> hi() {
        List<String> result = new ArrayList<>();
        result.add("hello spring boot");
    }
}
```

```

        result.add("hello docker");
        return result;
    }
}

```

接下来，我们通过如下命令编译和运行实例。

```
mvn package && java -jar target/spring-boot-docker-1.0.0.jar
```

服务实例启动后，我们使用 curl 通过默认端口请求资源路径（curl :8080/hi），得到如下测试结果。

```
["hello spring boot","hello docker"]
```

服务成功通过测试，接下来我们来为服务制作镜像。

2. 制作镜像

制作镜像是通过 docker build 命令完成的，Maven 的 Docker 插件 docker-maven-plugin 对其进行了封装，使得构建过程可以继构建成功后直接开始。配置该插件的示例如下。

```

<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.2.3</version>
  <configuration>
    <imageName>feuyeux/${project.artifactId}</imageName>
    <dockerDirectory>src/main/docker</dockerDirectory>
    <resources>
      <resource>
        <targetPath></targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>

```

上述配置示例中，imageName 中定义了镜像的名称，dockerDirectory 中定义的是 Dockerfile 所在的路径，示例中指定的是 src/main/docker。我们进入该目录，编辑 Dockerfile 文件。

```

FROM java:8
VOLUME /tmp
ADD spring-boot-docker-1.0.0.jar my_app.jar
RUN bash -c 'touch /my_app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/my_app.jar"]

```

在上述代码中，FROM java:8 是指镜像以 java:8 为基础，在其上生成新的层（Layer）。Docker 镜像的每一层都是只读的，上层可以复用下层的环境。VOLUME /tmp 是将容器内的 /tmp 目录映射到主机，使主机可以访问容器内的目录。ADD 是拷贝主机上的文件到容器

内，这里是将构建好的微服务 jar 拷贝到容器内的根目录，命名为 my_app.jar。RUN 用于定义在构建镜像时执行的命令。ENTRYPOINT 用于容器执行时运行的命令，类似的关键字是 CMD。一个 Dockerfile 内可以定义多个 CMD，但只能有一个 ENTRYPOINT。

Dockerfile 文件定义完毕，我们可以开始构建镜像了。在执行构建之前，确认开发环境中，启动了 Docker Engine。本示例运行在 Mac Book Pro 中，因此要先启动 Docker Toolbox 中的 Docker Quickstart Terminal.app 程序。本示例的完整示意如图 8-1 所示。

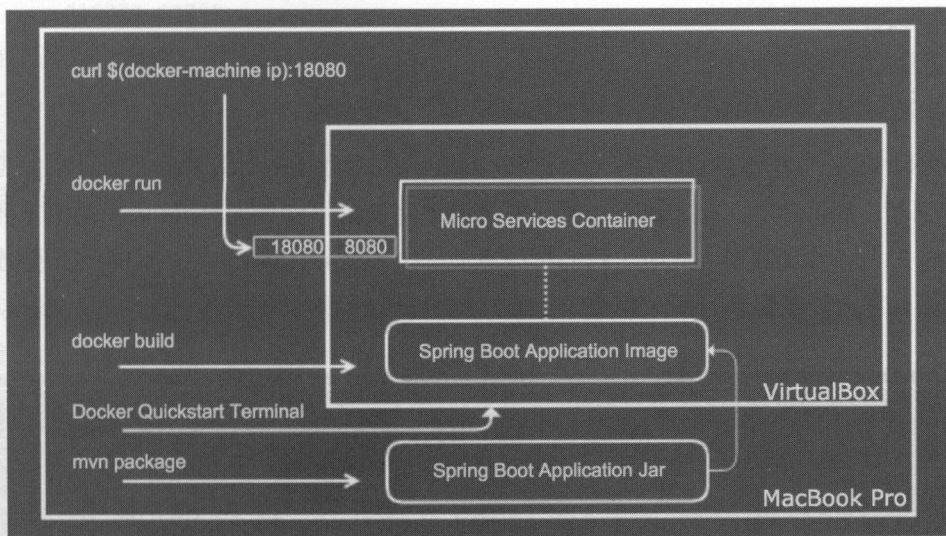


图 8-1 服务容器化实践环境示意图

如图 8-1 中，我们通过 mvn package 构建服务，生成 jar 包，然后通过 Dockerfile 中的定义，在执行构建镜像时，将 jar 拷贝到 Docker 镜像中。接下来，启动容器，暴露 18080 端口，在主机上使用 curl 命令访问校验。

接下来，我们就遵循上述过程，首先使用以下命令构建服务工程并构建镜像。

```
mvn package docker:build
```

构建成功后，可以通过 docker images feuyeux/spring-boot-docker 命令查看。

3. 运行容器

镜像可以分发到任何支持 Docker 的环境，然后我们使用如下命令运行容器。

```
docker run -p 18080:8080 -d feuyeux/spring-boot-docker
```

docker run 用于运行容器，参数 -p 是端口映射，容器中的微服务默认使用 8080，将其映射到主机的 18080 端口，参数 -d 是以守护进程（Daemon）的形式运行容器，最后是镜像

的名称。

在主机上使用如下命令测试容器中的微服务。

```
curl $(docker-machine ip):18080/hi  
["hello spring boot","hello docker"]
```

`docker-machine ip` 命令将输出 Virtualbox 实例的 IP 地址，即 Docker 所在的主机 IP。如果使用 Linux，Docker 所在的主机就是 Linux 系统，此处可以使用 `localhost`。

到此，我们完成了最基本的微服务容器化的工作。接下来，我们实现微服务自身有服务依赖的场景。

8.2.2 开发自测容器化

Docker 轻量化的容器技术对开发自测阶段的帮助显而易见。比如我们的微服务需要实现缓存机制，底层依赖 Redis 服务，但开发机上没有或者不希望安装 Redis，而测试环境又资源紧张，无法实现独享。此时，我们可以启动一个 Redis 容器，并让我们的微服务连接到该容器上，从而轻松实现自测过程中的依赖，如图 8-2 所示。

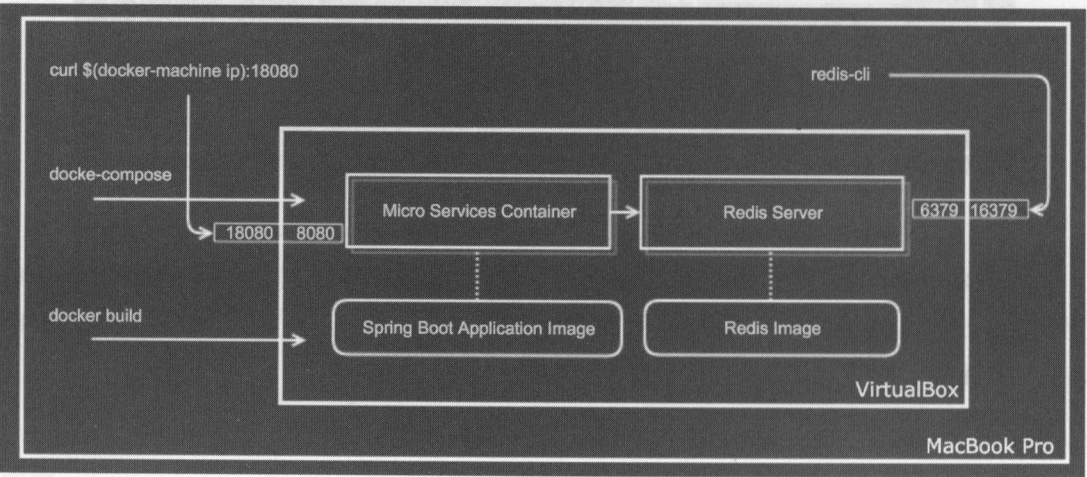


图 8-2 服务依赖容器化实践环境示意图

在图 8-2 中，在 8.2.1 节的基础上增加了 Redis 镜像，我们需要编排服务的关系，并暴露 Redis 服务端口给主机，以便通过 `redis-cli` 操作 Redis 服务。

1. 服务编排

我们使用 Docker Compose 实现服务的编排，配置示例如下。

```
nano docker-compose.yml
```



```

app:
  image: feuyeux/spring-boot-docker
  links:
    - redis
  ports:
    - "18080:8080"
redis:
  image: redis
  environment:
    - REDIS_PASS=cache_me
  ports:
    - "16379:6379"

```

在上述代码中存在两个容器，分别是 `app` 和 `redis`。`image` 中定义相应的镜像名称，`links` 指明了 `app` 要依赖 `redis` 容器，`ports` 中定义了端口映射。最后，`redis` 容器启动时需要密码，如果不指定将随机生成，因为我们的微服务要持久访问 `redis`，因此需要通过 `REDIS_PASS` 参数指定密码。

2. 缓存服务

接下来，我们回到微服务自身，讲述下如何实现缓存服务。

首先我们在 Spring Boot 工程中增加 `redis` 依赖，示例如下。

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-redis</artifactId>
  <version>${spring-boot.version}</version>
</dependency>

```

在配置文件 `application.properties` 中增加如下配置，其中 `192.168.99.100` 是 Docker Machine 默认的 IP。

```

spring.redis.database=0
spring.redis.host=192.168.99.100
spring.redis.password=cache_me
spring.redis.port=16379

```

缓存服务层的代码实现如下。

```

@Service
@CacheConfig(cacheNames = "saying")
public class CacheComponent {
    @Cacheable
    public String hi(String v) {
        return v + "-" + System.nanoTime();
    }
}

```

在上述代码中，`@CacheConfig` 定义了缓存主键，`@Cacheable` 指定需要缓存的方法。`hi` 方法每次返回系统当前时间的纳秒值，如果不缓存，每次结果肯定不同。

REST 服务的资源层将调用缓存服务，示例如下。

```
@Path("hi")
public class DemoResource {
    @Autowired
    private CacheComponent cacheComponent;

    @GET
    @Path("/{v}")
    public String hi2(@PathParam("v") String v) {
        return cacheComponent.hi(v);
    }
}
```

3. 启动服务

再次回到 Docker，与 8.2.1 节一样，使用 `mvn package docker:build` 命令构建镜像，然后在 `docker-compose.yml` 所在路径下，执行 `docker-compose up` 命令，启动全部容器。

多次执行 `curl $(docker-machine ip):18080/hi/hello123` 命令测试服务，将得到相同的结果，证明缓存生效。结果示例如下。

```
hello123-9453340548492
```

接下来，我们使用 Redis 客户端命令行测试服务。

```
redis-cli -a cache_me -h $(docker-machine ip) -p 16379

192.168.99.100:16379> keys *
1) "saying-keys"
2) "\xac\xed\x00\x05t\x00\bhello123"

192.168.99.100:16379> zrange "saying-keys" 0 -1
1) "\xac\xed\x00\x05t\x00\bhello123"

192.168.99.100:16379> get "\xac\xed\x00\x05t\x00\bhello123"
"\xac\xed\x00\x05t\x00\x16hello123-9453340548492"
```

上述示例中，缓存主键有两个，分别是 `"saying-keys"` 和 `"\xac\xed\x00\x05t\x00\bhello123"`，前者是一个有序集合，用于存储全部缓存的键及其活跃权重得分，后者是其中的一条缓存键。使用 `zrange "saying-keys" 0 -1` 命令访问前者的值，可以看到包含了其他全部缓存键，使用 `get "\xac\xed\x00\x05t\x00\bhello123"` 命令访问后者，可以得到与 `hi` 方法返回值相同的值。

8.3 容器化微服务

通过 8.2 节的阅读，我们掌握了如何将一个微服务及其依赖的服务容器化。本节将结合 7.3 节和 8.2 节的知识，在生产环境中完成一个高可用的微服务容器组，如图 8-3 所示。

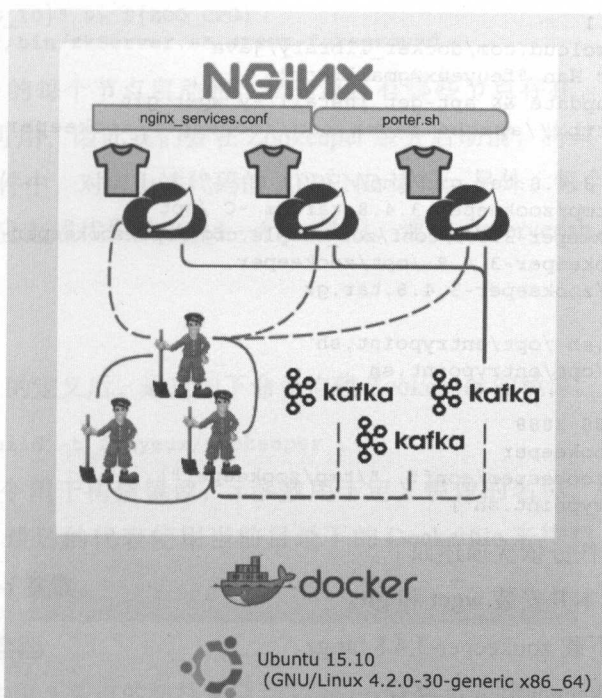


图 8-3 高可用服务容器化示意图

在图 8-3 中，我们生产环境的主机操作系统是 Ubuntu，其上 Docker Engine。7.3 节相关的全部服务（Nginx、微服务、Zookeeper）都以容器的形式运行在主机操作系统上，此外，本节增加了服务之间的消息流处理服务 Kafka。

阅读指南

8.3 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/8.3.boot.zk.kaka>。

8.3.1 Zookeeper

协调服务器 Zookeeper 的拓扑支持多节点在同一台主机部署，也支持每节点在单独一台主机部署。

1. Zookeeper 镜像制作

首先，我们定义 Dockerfile，制作 Zookeeper 镜像。使用 nano Dockerfile 命令编辑如下脚本。

```

# Version: 0.0.1
FROM index.tenxcloud.com/docker_library/java
MAINTAINER Eric Han "feuyeux@gmail.com"
# RUN apt-get update && apt-get install -y wget git
# RUN wget http://apache.fayea.com/zookeeper/zookeeper-3.4.8/zookeeper-3.4.8.tar.gz
COPY zookeeper-3.4.8.tar.gz /tmp/
RUN tar -xzf /tmp/zookeeper-3.4.8.tar.gz -C /opt
RUN cp /opt/zookeeper-3.4.8/conf/zoo_sample.cfg /opt/zookeeper-3.4.8/conf/zoo.cfg
RUN mv /opt/zookeeper-3.4.8 /opt/zookeeper
RUN rm -f /tmp/zookeeper-3.4.8.tar.gz

ADD entrypoint.sh /opt/entrypoint.sh
RUN chmod 777 /opt/entrypoint.sh

EXPOSE 2181 2888 3888
WORKDIR /opt/zookeeper
VOLUME ["/opt/zookeeper/conf", "/tmp/zookeeper"]
CMD ["/opt/entrypoint.sh"]

```

上述代码的主要任务依次描述如下。

- (1) 更新软件版本并安装 wget 和 git。
- (2) 使用 wget 下载 zookeeper-3.4.8.tar.gz。
- (3) 解压缩 Zookeeper 到 /opt 路径。
- (4) 拷贝配置文件 zoo_sample.cfg 为 zoo.cfg。
- (5) 将容器启动脚本 entrypoint.sh 拷贝到 /opt/ 并修改为可执行。
- (6) 开放 2181、2888、3888 端口。
- (7) 设定容器的当前目录为 /opt/zookeeper。
- (8) 将容器目录 /opt/zookeeper/conf 映射到主机目录 /tmp/zookeeper。
- (9) 定义容器启动时，执行 /opt/entrypoint.sh 脚本。

2. 容器启动脚本

接下来，使用 nano entrypoint.sh 命令编辑如下脚本。

```

#!/bin/sh
ZOO_CFG="/opt/zookeeper/conf/zoo.cfg"
echo "server id (myid): ${SERVER_ID}"
echo "${SERVER_ID}" > /tmp/zookeeper/myid

echo "${APPEND_1}" >> ${ZOO_CFG}
echo "${APPEND_2}" >> ${ZOO_CFG}
echo "${APPEND_3}" >> ${ZOO_CFG}
echo "${APPEND_4}" >> ${ZOO_CFG}
echo "${APPEND_5}" >> ${ZOO_CFG}
echo "${APPEND_6}" >> ${ZOO_CFG}
echo "${APPEND_7}" >> ${ZOO_CFG}
echo "${APPEND_8}" >> ${ZOO_CFG}
echo "${APPEND_9}" >> ${ZOO_CFG}

```

```
echo "${APPEND_10}" >> ${ZOO_CFG}
/opt/zookeeper/bin/zkServer.sh start-foreground
```

因为 Zookeeper 的每个节点启动时，要知道还有哪些节点存在，以便完成 ZK 集群自身的主从选举和高可用，因此我们要在 Zookeeper 服务启动前，将其他节点的配置追加到 Zookeeper 的配置文件中，对应上述代码的 APPEND 诸行。另外，每个节点要分配一个唯一的 id 作为标识，对应上述代码的 myid 一行。最后，通过 zkServer.sh 脚本以前台运行的方式启动服务。

3. 构建镜像

完成 Dockerfile 的定义后，通过如下命令构建 Zookeeper 的镜像。

```
sudo docker build -t feuyeux/zookeeper .
```

docker build 命令用于构建镜像，-t 参数用于定义镜像的名称，接下来是镜像的名称 feuyeux/zookeeper，最后的代表使用当前目录下的 Dockerfile 来构建。如果需要特殊指定 Dockerfile，请使用 -f 参数。

4. 单主机启动容器

我们首先以单主机 3 节点的拓扑来启动 Zookeeper 容器集群。

```
HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\\d\\.]+)\\/\\d+\\s}) { print $1 }')
```

```
sudo docker run -d \
--name=zk1 \
--net=host \
-e SERVER_ID=1 \
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
-e APPEND_4=clientPort=2181 \
feuyeux/zookeeper
```

```
sudo docker run -d \
--name=zk2 \
--net=host \
-e SERVER_ID=2 \
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
-e APPEND_4=clientPort=2182 \
feuyeux/zookeeper
```

```
sudo docker run -d \
--name=zk3 \
--net=host \
-e SERVER_ID=3 \
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
```



```
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
-e APPEND_4=clientPort=2183 \
feuyeux/zookeeper
```

在上述脚本中，首先获取主机 eth0 的 IP 地址，然后分别启动 3 个 Zookeeper 节点，每个节点容器有自己的名字（通过 --name 指定），myid 的值通过 SERVER_ID 行指定。--net=host 表示容器使用主机网络，每个节点通过相同的 IP 不同的端口号启动，对外服务的端口号依次为 2181 ~ 2183，对内通信的端口依次为 2888 ~ 2890 和 3888 ~ 3890。

容器启动后，可以通过如下命令观察容器配置文件。

```
sudo docker exec -ti zk1 cat /opt/zookeeper/conf/zoo.cfg
sudo docker exec -ti zk1 cat /tmp/zookeeper/myid
```

上述脚本使用 docker exec 命令，访问名为 zk1 的容器的信息。需要注意的是，docker exec 命令的目的是观察容器，根据容器文化，我们不要使用这个命令去修改容器的状态。

在主机上，可以使用 zkCli 命令测试 Zookeeper 集群。

```
HOST_IP=192.168.3.103
zkCli -server ${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183
```

在主机上，可以使用如下代码测试 Zookeeper 服务。

```
public interface Conf {
    String zkConnect = "192.168.3.103:2181,192.168.3.103:2182,192.168.3.103:2183";
    String kafkaServerList = "192.168.3.103:9091,192.168.3.103:9092,192.168.3.103:9093";
    String topic = "ktalk";
    int zkTimeout = 10000;
}

public static void main(String[] args) throws Exception {
    // 关注点 1
    ZooKeeper zk = new ZooKeeper(Conf.zkConnect, Conf.zkTimeout, new Watcher() {
        public void process(WatchedEvent event) {
            System.out.println("Event:" + event.getType());
        }
    });
    // 关注点 2
    zk.create("/my_path", "root_data".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE,
        CreateMode.PERSISTENT);
    zk.create("/my_path/my_branch", "branch_data".getBytes(), ZooDefs.Ids.OPEN_
        ACL_UNSAFE, CreateMode.PERSISTENT);

    System.out.println("my_path data:" + new String(zk.getData("/my_path",
        false, null)));
    System.out.println("my_branch data:" + new String(zk.getData("/my_path/my_
        branch", false, null)));
    System.out.println("my_path path:" + zk.getChildren("/my_path", true));
    // 关注点 3
    zk.setData("/my_path/my_branch", "branch_new_data".getBytes(), -1);
```

```

    System.out.println("my_branch data:" + new String(zk.getData("/my_path/my_
branch", false, null)));
// 关注点 4
    zk.delete("/my_path/my_branch", -1);
    System.out.println("my_path path:" + zk.getChildren("/my_path", true));
    zk.delete("/my_path", -1);
}

```

在上述代码中，使用 Conf 中的配置启动 Zookeeper 实例，见关注点 1；接下来，创建持久化路径 my_path 和 /my_path/my_branch，分别赋值为 root_data 和 branch_data，见关注点 2；然后，更新 /my_path/my_branch 路径的值为 branch_new_data，见关注点 3；最后，删除路径 branch_new_data。全部的输出示例如下。

```

Event:None
my_path data:root_data
my_branch data:branch_data
my_path path:[my_branch]
my_branch data:branch_new_data
Event:NodeChildrenChanged
my_path path:[]
Event:NodeDeleted

```

5. 多主机启动容器

单主机存在单点故障的风险，我们可以使用相同的方式，将 3 个 Zookeeper 容器分别在 3 台主机上启动，示例如下。

```

docker run -d \
--net=host \
-e SERVER_ID=1 \
-e APPEND_1=server.1=zk1:2888:3888 \
-e APPEND_2=server.2=zk2:2888:3888 \
-e APPEND_3=server.3=zk3:2888:3888 \
feueux/zookeeper

```

```

docker run -d \
--net=host \
-e SERVER_ID=2 \
-e APPEND_1=server.1=zk1:2888:3888 \
-e APPEND_2=server.2=zk2:2888:3888 \
-e APPEND_3=server.3=zk3:2888:3888 \
feueux/zookeeper

```

```

docker run -d \
--net=host \
-e SERVER_ID=3 \
-e APPEND_1=server.1=zk1:2888:3888 \
-e APPEND_2=server.2=zk2:2888:3888 \
-e APPEND_3=server.3=zk3:2888:3888 \
feueux/zookeeper

```

上述代码中，zk1 为第一台主机的 hostname 或者 IP。多主机的情况下，每个 Zookeeper

的端口都使用默认的。

Zookeeper 服务除了 7.3 节中讲述的服务注册和服务发现，还将用于保证 Kafka 集群的高可用。接下来，我们实现 Kafka 集群的容器化。

8.3.2 Kafka

Kafka 是微服务、大数据领域的主要流处理服务产品。Kafka 容器的拓扑相对于 Zookeeper 要简单，只需暴露一个服务端口即可。本例所使用的 Zookeeper 集群的拓扑采用单主机方式，与多主机方式非常类似，改动很小。

1. Kafka 镜像制作

首先使用 nano Dockerfile 命令编辑如下脚本。

```
# Version: 0.0.1
FROM index.tenxcloud.com/docker_library/java
MAINTAINER Eric Han "feuyeux@gmail.com"
#RUN apt-get update
RUN apt-get install -y wget git
RUN wget -q http://apache.fayea.com/kafka/0.9.0.1/kafka_2.10-0.9.0.1.tgz
RUN tar -xzf kafka_2.10-0.9.0.1.tgz -C /opt
RUN mv /opt/kafka_2.10-0.9.0.1 /opt/kafka
ENV KAFKA_HOME /opt/kafka
ADD start-kafka.sh /usr/bin/start-kafka.sh
RUN chmod 777 /usr/bin/start-kafka.sh
CMD start-kafka.sh
```

上述代码的主要任务依次描述如下。

- (1) 更新软件版本并安装 wget 和 git。
- (2) 使用 wget 下载 kafka_2.10-0.9.0.1.tgz。
- (3) 解压缩 Kafka 到 /opt 路径。
- (4) 定义 Kafka 主目录环境变量 KAFKA_HOME。
- (5) 将容器启动脚本拷贝为 /usr/bin/start-kafka.sh 并修改为可执行。
- (6) 定义容器启动时，执行 start-kafka.sh 脚本。

2. 容器启动脚本

使用 nano start-kafka.sh 命令编辑如下脚本。

```
cp $KAFKA_HOME/config/server.properties $KAFKA_HOME/config/server.properties.bk
sed -r -i "s/(zookeeper.connect)=(.*)/\1=${ZK}/g" $KAFKA_HOME/config/server.properties
sed -r -i "s/(broker.id)=(.*)/\1=${BROKER_ID}/g" $KAFKA_HOME/config/server.properties
sed -r -i "s/(log.dirs)=(.*)/\1=/tmp/kafka-logs-${BROKER_ID}/g" $KAFKA_HOME/config/server.properties
```

```
sed -r -i "s/#(advertised.host.name)=(.*)/\1=${HOST_IP}/g" $KAFKA_HOME/config/
server.properties
sed -r -i "s/#(port)=(.*)/\1=${PORT}/g" $KAFKA_HOME/config/server.properties
sed -r -i "s/(listeners)=(.*)/\1=PLAINTEXT:///:${PORT}/g" $KAFKA_HOME/config/
server.properties
if [ "$KAFKA_HEAP_OPTS" != "" ]; then
    sed -r -i "s/^(export KAFKA_HEAP_OPTS)=\"(.*)\"/\1=\"${KAFKA_HEAP_OPTS}\"/g"
    $KAFKA_HOME/bin/kafka-server-start.sh
fi
$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties
```

启动脚本的主要目的是动态配置 Zookeeper 和自身服务的 IP 和端口。sed 工具通过正则表达式匹配配置文件 server.properties 中要替换的项，使用启动容器时传入的参数进行替换。

3. 构建镜像

完成 Dockerfile 的定义后，通过如下命令构建 Kafka 的镜像。

```
sudo docker build -t feuyeux/kafka .
```

4. 启动容器

使用 feuyeux/kafka 镜像，启动 3 个 Kafka 容器实例。

```
HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\d\.]+)\s/\d+/\s}
xms) { print $1 }')
```

```
K_PORT=9091
```

```
sudo docker run --name=k1 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=1 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,$ {HOST_IP}:2182,$ {HOST_IP}:2183 \
-d feuyeux/kafka
```

```
K_PORT=9092
```

```
sudo docker run --name=k2 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=2 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,$ {HOST_IP}:2182,$ {HOST_IP}:2183 \
-d feuyeux/kafka
```

```
K_PORT=9093
```

```
sudo docker run --name=k3 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=3 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,$ {HOST_IP}:2182,$ {HOST_IP}:2183 \
-d feuyeux/kafka
```

上述脚本中，K_PORT 变量用于定义当前容器中 Kafka 服务的端口，-e 参数中定义的

变量都是启动脚本中使用的，请结合 start-kafka.sh 脚本理解。

容器启动后，我们通过如下命令观察容器配置文件。

```
sudo docker exec -ti k1 cat /opt/kafka/config/server.properties
```

8.3.3 微服务

有了 Zookeeper 和 Kafka 容器集群，我们就可以基于它们编写自己的微服务了。

首先，我们根据 7.3.1 节的服务发现示例，编写入口类 KafkaApplication，示例代码如下。

```
@SpringBootApplication
@EnableDiscoveryClient
@PropertySource("file:/opt/zk.kaka.properties")
public class KafkaApplication {
    public static void main(String[] args) {
        SpringApplication.run(KafkaApplication.class, args);
    }
}
```

上述代码中，服务的配置文件并不存在于 jar 中，而是定义在 /opt/zk.kaka.properties 中，目的是可以动态修改配置，而不改变代码。

1. Kafka 读写

资源层在 7.3.1 节示例的基础上，为 hi 方法增加了使用 Kafka 记录访问日志的功能，增加了 kk 方法获取访问日志，示例如下。

```
@Path("/kk")
@GET
@Produces("application/json")
public List<String> k() {
    return EagleService.getTemp();
}

@Path("/hi")
@GET
public String hi() {
    ServiceInstance serviceInstance = discovery.getLocalServiceInstance();
    String result = serviceInstance.getHost() + ":" + serviceInstance.getPort();
    if (request != null) {
        dogService.process(request.getRemoteAddr());
    }
    return result;
}
```

在服务层中，新增 DogService 类生产日志信息，新增 EagleService 类消费日志信息，示例如下。


```

@Service
public class DogService {
    @Value("${kafkaServerList}")
    private String kafkaServerList;
    @Value("${topic}")
    private String topic;
    @Value("${key}")
    private String key;

    public void process(String value) {
        if (isAsync) {
            producer.send(new ProducerRecord<>(topic, value), new KafCallBack(value));
        }
        ...
    }
}

@Service
public class `EagleService` extends ShutdownableThread {
    @Value("${group}")
    private String group;
    @Value("${topic}")
    private String topic;
    @Value("${kafkaServerList}")
    private String kafkaServerList;
    private static List<String> temp = new ArrayList<>();

    @Override
    public void doWork() {
        consumer.subscribe(Collections.singletonList(this.topic));
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records) {
            temp.add(record.key() + ", " + record.value() + " at offset " +
record.offset());
        }
    }

    public static List<String> getTemp() {
        return temp;
    }
}

```

2. 制作镜像

介绍完微服务的功能，我们将这个服务容器化。首先，我们考虑如何编写启动脚本。

使用 nano start.sh 编辑启动脚本如下。

```

#!/usr/bin/env bash
echo "ZK=${ZK}"
echo "KAFKA=${KAFKA}"
sed -r -i "s/(spring.cloud.zookeeper.connectString)=(.*)/\1=${ZK}/g" /opt/
zk.kaka.properties
sed -r -i "s/(zkConnect)=(.*)/\1=${ZK}/g" /opt/zk.kaka.properties;
sed -r -i "s/(kafkaServerList)=(.*)/\1=${KAFKA}/g" /opt/zk.kaka.properties
echo "zk.kaka.properties:"

```

```

echo "===="
cat /opt/zk.kaka.properties
echo "===="
echo "start micro services..."
java -Djava.security.egd=file:/dev/./urandom -Dspring.cloud.bootstrap.
location=/opt/zk.kaka.properties -jar /my_app.jar

```

上述脚本动态修改配置文件 /opt/zk.kaka.properties，然后在启动微服务的过程中，通过 spring.cloud.bootstrap.location 参数为 Spring Cloud 编写的服务指定 bootstrap 配置文件，以便服务发现等功能生效。

接下来，编写 Dockerfile 文件，将启动脚本和配置文件拷贝到镜像中。

```

#FROM java:8
VOLUME /tmp
ADD boot.zk.kaka-0.0.1-SNAPSHOT.jar my_app.jar
ADD zk.kaka.properties /opt/zk.kaka.properties
ADD start_services.sh start_services.sh
RUN chmod 777 start_services.sh
RUN bash -c 'touch /my_app.jar'
ENTRYPOINT ["/start.sh"]

```

使用如下命令构建服务，并将服务 jar 包拷贝到构建镜像的 Linux 环境中。

```

mvn clean package -DskipTests
scp target/boot.zk.kaka-0.0.1-SNAPSHOT.jar han@10.211.55.32:/home/han/zk.kk

```

在 Linux 环境中，执行如下命令构建微服务镜像。

```

sudo docker build -t feuyeux/boot.zk.kaka

```

3. 运行容器

使用微服务镜像 feuyeux/boot.zk.kaka 启动 3 个实例。

```

HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\d.]+)\s}\s){ print $1 }')
PORT=18080
sudo docker run -d \
--name=kaka1 \
--hostname=kaka1 \
-p ${PORT}:8080 \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
feuyeux/boot.zk.kaka

PORT=18081
sudo docker run -d \
--name=kaka2 \
--hostname=kaka2 \
-p ${PORT}:8080 \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
feuyeux/boot.zk.kaka

```

```

PORT=18082
sudo docker run -d \
--name=kaka3 \
--hostname=kaka3 \
-p ${PORT}:8080 \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
feuyeux/boot.zk.kaka

```

然后分别测试 3 个实例。

```

han@Ubuntu1510:~$ http --body localhost:18080/hi
172.17.0.5:8080

han@Ubuntu1510:~$ http --body localhost:18081/hi
172.17.0.6:8080

han@Ubuntu1510:~$ http --body localhost:18082/hi
172.17.0.7:8080

```

我们得到的 IP 是 Docker 为每个容器动态分配的 Docker 网段的 IP，对外我们分别通过 Linux 主机 IP 和 18080 ~ 18082 端口访问。

接下来测试 NGINX 配置文件格式的注册服务列表。

```

han@Ubuntu1510:~$ http --body localhost:18082/hi2
server 172.17.0.7:8080;server 172.17.0.5:8080;server 172.17.0.6:8080;

```

结果看上去很熟悉，它和 7.3.1 节非常类似。

8.3.4 Nginx

接下来，我们将 Nginx 容器化。

1. 配置模板

首先使用 nano nginx_template.conf 命令编辑配置文件模板。

```

worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream my_service {
        MICRO_SERVICES
    }

    server {
        listen 8000;
        server_name localhost;
        location / {

```

```

proxy_pass http://my_service;
proxy_redirect off;
}
}
}

```

2. 动态修改模板

然后使用 nano porter.sh 编辑动态修改这个模板文件的脚本。

```

#!/usr/bin/env bash
port=18080
for (( ; ${port} <= 18090; port++ ))
do
    status=$(curl -s -w %{http_code} -o /dev/null localhost:${port}/hi2)
    echo "status=${status}"
    if [ ${status} == 200 ]; then
        break
    fi
done
if [ ${status} == 200 ]; then
    echo "port=${port}"
    cp nginx_template.conf /tmp/nginx/nginx.conf
    services=$(curl -s localhost:${port}/hi2)
    echo "services=${services}"
    sed -i -e "s/MICRO_SERVICES/${services}/g" /tmp/nginx/nginx.conf
    echo "Done"
else
    echo "No available services"
fi

```

这里之所以将最终的配置文件定义为 /tmp/nginx/nginx.conf, 是因为目录 /tmp/nginx 将与容器内的 Nginx 配置目录映射。

3. 基本配置

接着使用如下命令, 编辑基本配置文件。

```

mkdir /tmp/nginx
./porter.sh
cat /tmp/nginx/nginx.conf

```

配置文件内容如下。

```

worker_processes 2;

events {
    worker_connections 1024;
}

http {
    upstream my_service {
        server 172.17.0.7:8080;server 172.17.0.5:8080;server 172.17.0.6:8080;
    }
}

```

```

server {
    listen 8000;
    server_name localhost;
    location / {
        proxy_pass http://my_service;
        proxy_redirect off;
    }
}

```

4. 构建镜像

有了上述 3 个文件，我们就可以启动 Nginx 容器了。我们要将 Nginx 的默认静态文件目录 `usr/share/nginx/html` 和配置文件目录 `/etc/nginx` 通过 `-v` 映射到主机上，示例如下。

```

sudo docker run -d \
--name kaka-ng \
-v /tmp/website:/usr/share/nginx/html \
-v /tmp/nginx:/etc/nginx:ro \
-p 18000:8000 \
nginx

```

这里，我们并没有自己定义 `Dockerfile` 来构建镜像，而是直接使用官方的 Nginx 镜像启动的容器。

在主机上，使用如下命令测试 Nginx 容器，可以看到轮询请求可用的微服务节点。

```

han@Ubuntu1510:~/nginx$ http --body localhost:18000/hi
172.17.0.5:8080
han@Ubuntu1510:~/nginx$ http --body localhost:18000/hi
172.17.0.6:8080
han@Ubuntu1510:~/nginx$ http --body localhost:18000/hi
172.17.0.7:8080

```

同样，我们可以使用容器中的 `zkCli.sh` 来查看 Zookeeper 注册服务，还记得 8.1 节所述的可执行容器吗？

```

HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\d\.]+\s)/\s+\s+}) { print $1 }')

sudo docker run -ti --rm feuyex/zookeeper /opt/zookeeper/bin/zkCli.sh -server
${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183

ls /services/bootZKkafka

[35accd38-77c1-45c5-837f-a1f4833dfdl4, f3e4d59c-3cf4-49af-9f85-27ea889f7f76,
db62a968-237d-4921-800a-aed7caa70708]

```

使用容器内的 `kafka-topics.sh` 查看 kafka 的 topic。


```
HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\\d\\.]+)\\/\\d+\\s}
xms) { print $1 }')
sudo docker run -ti --rm feueux/kafka /opt/kafka/bin/kafka-topics.sh --list
--zookeeper ${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183
```

```
__consumer_offsets
```

```
my_story
```

这里包括了我们的微服务所使用的 my_story。使用容器内的 kafka-console-consumer.sh 查看 kafka 中 my_story 的全部记录。

```
sudo docker run -ti --rm feueux/kafka /opt/kafka/bin/kafka-console-consumer.
sh --zookeeper ${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 --from-
beginning --topic my_story
```

8.4 本章小结

本章以微服务的容器化为出发点，分3节讲述了容器、微服务容器化和高可用集群容器的实现。

最后，集群容器编排的脚本实例。

```
nano orchestration.sh
```

```
echo "stop and clean container, and remove unused images..."
```

```
sudo docker kill $(sudo docker ps -q)
```

```
sudo docker rm $(sudo docker ps -a -q)
```

```
sudo docker images | grep "<none>" | awk '{print $3}' | xargs sudo docker rmi
```

```
HOST_IP=$(ip -o -4 addr list eth0 | perl -n -e 'if (m{inet\s([\\d\\.]+)\\/\\d+\\s}
xms) { print $1 }')
```

```
echo "done."
```

```
sleep 1
```

```
echo "launch zookeeper cluster..."
```

```
sudo docker run -d \
```

```
--name=zkl \
```

```
--net=host \
```

```
-e SERVER_ID=1 \
```

```
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
```

```
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
```

```
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
```

```
-e APPEND_4=clientPort=2181 \
```

```
feueux/zookeeper
```

```
sudo docker run -d \
```

```
--name=zkl \
```

```
--net=host \
```

```
-e SERVER_ID=2 \
```

```
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
```

```
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
-e APPEND_4=clientPort=2182 \
feuyeux/zookeeper
```

```
sudo docker run -d \
--name=zk3 \
--net=host \
-e SERVER_ID=3 \
-e APPEND_1=server.1=${HOST_IP}:2888:3888 \
-e APPEND_2=server.2=${HOST_IP}:2889:3889 \
-e APPEND_3=server.3=${HOST_IP}:2890:3890 \
-e APPEND_4=clientPort=2183 \
feuyeux/zookeeper
```

```
echo "done"
```

```
echo "launch kafka cluster..."
```

```
K_PORT=9091
```

```
sudo docker run --name=k1 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=1 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-d feuyeux/kafka
```

```
K_PORT=9092
```

```
sudo docker run --name=k2 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=2 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-d feuyeux/kafka
```

```
K_PORT=9093
```

```
sudo docker run --name=k3 \
-p ${K_PORT}:${K_PORT} \
-e BROKER_ID=3 \
-e HOST_IP=${HOST_IP} \
-e PORT=${K_PORT} \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-d feuyeux/kafka
```

```
echo "zk and kafka started successfully."
```

```
echo "now, start micro services instance"
```

```
PORT=18080
```

```
sudo docker run -d \
--name=kakal \
--hostname=kakal \
-p ${PORT}:8080 \
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
```

```
feuyeux/boot.zk.kaka
```

```
PORT=18081
```

```
sudo docker run -d \
```

```
--name=kaka2 \
```

```
--hostname=kaka2 \
```

```
-p ${PORT}:8080 \
```

```
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
```

```
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
```

```
feuyeux/boot.zk.kaka
```

```
PORT=18082
```

```
sudo docker run -d \
```

```
--name=kaka3 \
```

```
--hostname=kaka3 \
```

```
-p ${PORT}:8080 \
```

```
-e ZK=${HOST_IP}:2181,${HOST_IP}:2182,${HOST_IP}:2183 \
```

```
-e KAFKA=${HOST_IP}:9091,${HOST_IP}:9092,${HOST_IP}:9093 \
```

```
feuyeux/boot.zk.kaka
```

```
sudo docker ps
```

JAX-RS 调优

基于 JAX-RS 的项目是部署在 Servlet 容器或者 Java EE 容器的 Web 服务，因此要提高这类项目的性能，除了在编码阶段需要注意代码的质量，在配置部署阶段还要关注于网络负载、Jersey 参数配置以及对 Java 虚拟机调优，在运行阶段还要对运行平台的操作系统、服务器软件进行监控和调优。本章将讲述如何为基于 JAX-RS 的项目进行调优。

阅读指南

本章示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/9.simple-service>。

9.1 使用缓存优化负载

系统的网络吞吐能力是其性能的重要指标，我们可以使用操作系统自带的工具进行监控。但监控只能判断网络的负载情况，而降低网络负载是提高网络吞吐能力的一个有效的手段。使用动静分离和缓存是降低系统和网络负载的手段之一。本节将讲述基于 JAX-RS 的项目的缓存实践。

9.1.1 缓存协商

Jersey 支持使用通用的 HTTP 头进行缓存协商。在 HTTP 头字段中，Expires 和 Cache-

Control、Last-Modified 用于作为缓存过期时间的判断依据。ETag 用于标识一次请求。If-Modified-Since 和 If-None-Match 是缓存控制的两种处理标识。如下分别讲述它们的使用。

❑ Expires：用于记录 HTTP 缓存过期时间头信息。Expires 简单易用，服务器据此判断请求是否应该刷新。但是 Expires 的标识比较粗糙，其缺点是当服务器的时区和本地时间与客户端上的不一致时，会影响缓存的准确性。

❑ Cache-Control：用于控制 HTTP 缓存属性的头信息。常用的属性包括：

- Public：响应可缓存在任何缓存区。
- Private：响应只能缓存在私有缓存区，不能被共享缓存处理。
- no-cache：响应不能缓存（HTTP/1.0 用 Pragma 的 no-cache 替换）。
- max-age：缓存最大时间（以秒为单位）。和 Expires 共存时，优先使用 max-age。

和 Expires 对比，max-age 使用的是相对时间，因此不存在 Expires 的问题。

❑ Last-Modified：页面文件最后修改时间，精确到秒级。

❑ If-Modified-Since：是指自从某个时间点开始服务器提供了最新数据。当服务器从客户端的请求中读出这个标识后，结合 Cache-Control 中定义的缓存参数，判断是否需要刷新数据来响应当前请求。如图 9-1 所示，客户端请求一个 REST 资源，服务器返回最新数据并指定 Cache-Control 的 max-age 为 1200 秒，Last-Modified 为当前时间。之后客户端再次发起对该资源的请求，服务器根据 Cache-Control 的 max-age 和 Last-Modified 计算是否需要刷新数据。如果需要，即执行业务处理并返回最新的数据，HTTP 状态码为 200 OK，否则直接返回缓存数据，HTTP 状态码为 304 Not Modified。

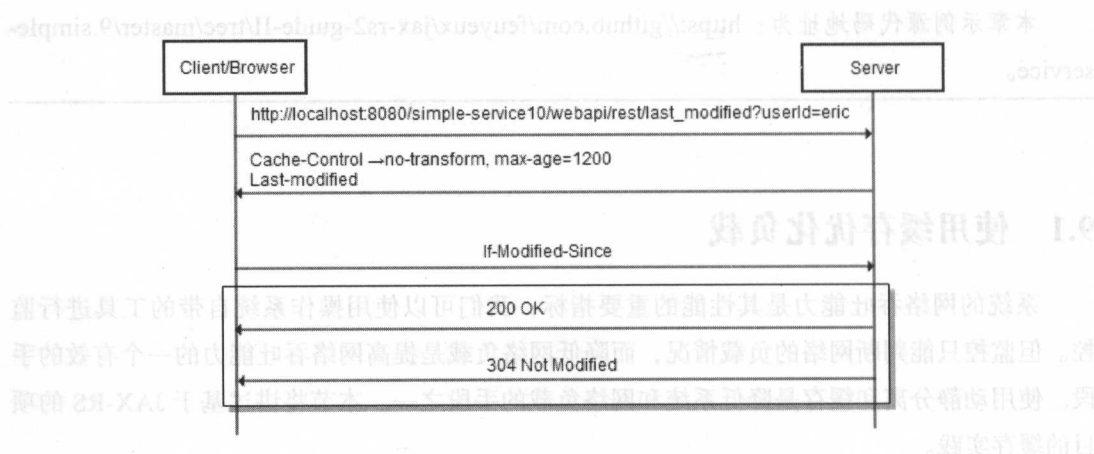


图 9-1 If-Modified-Since 流程示意图

❑ ETag 服务器端的散列值，每次请求处理都是唯一的。

❑ If-None-Match：是指请求的 ETag 是否匹配，如图 9-2 所示，当服务器从客户端的请

求中读出这个标识后，会与当前 ETag 对比，如果符合即返回缓存数据，HTTP 状态码为 304Not Modified，否则返回最新数据，HTTP 状态码为 200 OK。

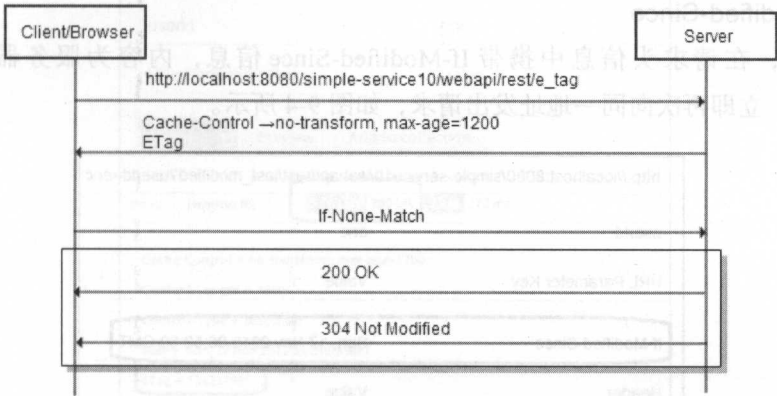


图 9-2 If-None-Match 流程示意图

9.1.2 条件 GET

条件 GET 是使 If-Modified-Since 或者 If-None-Match 头信息对 GET 请求进行缓存处理的一种方案。本节使用本章提供的示例项目演示如何实现对条件 GET 方法的支持。

1. Last-Modified

使用 Chrome 浏览器的 POSTMAN 插件向支持 If-Modified-Since 条件 GET 请求的 REST 资源发起请求，地址为：`http://localhost:8080/simple-service/webapi/rest/last_modified`，如图 9-3 所示。

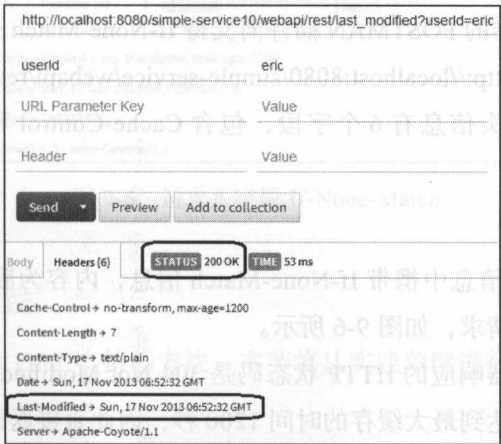


图 9-3 响应头字段 Last-Modified

在图 9-3 中, 返回头信息有 6 个字段, 包含 Cache-Control 和 Last-Modified 信息, 同时 HTTP 状态码为 200 OK。

2. If-Modified-Since

接下来, 在请求头信息中携带 If-Modified-Since 信息, 内容为服务器返回的 Last-Modified 值。立即再次向同一地址发出请求, 如图 9-4 所示。

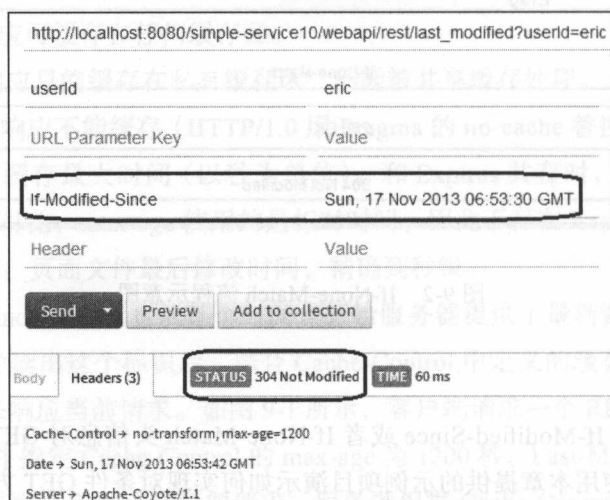


图 9-4 请求头字段 If-Modified-Since

在图 9-4 中, 服务器响应的 HTTP 状态码是 304 Not Modified, 因为最后修改时间与服务器当前时间之差没有达到最大缓存的时间 1200 秒, 因此直接返回缓存数据。

3. ETag

使用 Chrome 浏览器的 POSTMAN 插件向支持 If-None-Match 条件 GET 请求的 REST 资源发起请求, 地址为: http://localhost:8080/simple-service/webapi/rest/e_tag, 如图 9-5 所示。

在图 9-5 中, 返回头信息有 6 个字段, 包含 Cache-Control 和 ETag 信息, 同时 HTTP 状态码为 200 OK。

4. If-None-Match

接下来, 在请求头信息中携带 If-None-Match 信息, 内容为服务器返回的 ETag 值。立即再次向同一地址发出请求, 如图 9-6 所示。

在图 9-6 中, 服务器响应的 HTTP 状态码是 304 Not Modified, 因为最后修改时间与服务器当前时间之差没有达到最大缓存的时间 1200 秒, 因此直接返回缓存数据。

通过上述的实验, 我们从浏览器端了解了 HTTP 条件 GET 的原理及其在 REST 服务中

的实践。接下来，我们走进服务器端，讲述 REST 缓存的代码实现。

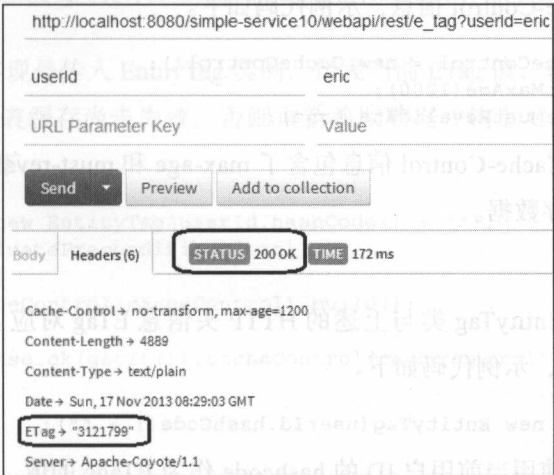


图 9-5 响应头字段 ETag

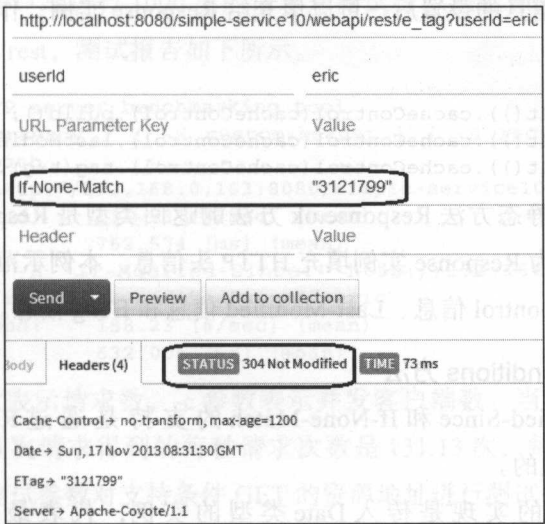


图 9-6 请求头字段 If-None-Match

9.1.3 REST 缓存实践

Jersey 提供了对条件 GET 支持的方法，本节将从实践角度进行讲述。

1. CacheControl

JAX-RS 提供了 CacheControl 类与上述的 HTTP 头信息 Cache-Control 对应。在 REST

请求处理的方法中，可以通过构造 `CacheControl` 实例，并为其设置参数值，使其作为 HTTP 响应头信息中的 `Cache-Control` 信息，示例代码如下。

```
CacheControl cacheControl = new CacheControl();
cacheControl.setMaxAge(1200);
cacheControl.setMustRevalidate(true);
```

在这段代码中，`Cache-Control` 信息包含了 `max-age` 和 `must-revalidate`，分别设置为 2 分钟和强制客户端不缓存数据。

2. EntityTag

JAX-RS 提供了 `EntityTag` 类与上述的 HTTP 头信息 `ETag` 对应。`EntityTag` 实例用来设置 HTTP 头信息 `ETag`，示例代码如下。

```
EntityTag tag = new EntityTag(userId.hashCode() + "");
```

在这段代码中，使用当前用户 ID 的 `hashCode` 作为 `ETag` 的值。

3. ResponseBuilder

在完成 HTTP 头信息的设置后，可以填充给 `Response` 实例，作为响应头信息，示例代码如下。

```
Response.ok(getIt()).cacheControl(cacheControl).build();
Response.ok(getIt()).cacheControl(cacheControl).lastModified(date).build();
Response.ok(getIt()).cacheControl(cacheControl).tag(tag).build();
```

在这段代码中，静态方法 `Response.ok` 方法的返回类型是 `ResponseBuilder`，通过使用 `ResponseBuilder` 可以为 `Response` 实例填充 HTTP 头信息。本例示范了不同 HTTP 头信息的填充，分别是 `Cache-Control` 信息、`Last-Modified` 信息和 `ETag` 信息。

4. evaluatePreconditions 方法

Jersey 对 `If-Modified-Since` 和 `If-None-Match` 的支持是通过 `Request` 接口的 `evaluatePreconditions` 方法实现的。

`If-Modified-Since` 的实现是传入 `Date` 类型的实例，代表最后修改时间，输出是 `ResponseBuilder` 实例，如果不为空即代表缓存尚未失效。否则重新获取数据并构造 `Cache-Control` 和 `lastModified` 响应头信息，示例代码如下。

```
Date lastModified = map.get(userId);
Response.ResponseBuilder rb = null;
if (lastModified != null) {
    rb = request.evaluatePreconditions(lastModified);
}
if (rb != null) {
    return rb.cacheControl(cacheControl).build();
} else {
```

```

    Date date = new Date();
    map.put(userId, date);
    return Response.ok(getIt()).cacheControl(cacheControl).lastModified(date).build();
}

```

If-None-Match 的实现是传入 EntityTag 实例，代表当前 ETag 值，输出是 ResponseBuilder 实例，如果不为空即代表缓存尚未失效。否则重新获取数据并构造 Cache-Control 和 ETag 响应头信息，示例代码如下。

```

EntityTag tag = new EntityTag(userId.hashCode() + "");
rb = request.evaluatePreconditions(tag);
if (rb != null) {
    returnrb.cacheControl(cacheControl).build();
} else {
    return Response.ok(getIt()).cacheControl(cacheControl).tag(tag).build();
}

```

9.1.4 ab 测试

ab 是 Apache 的 HTTP 服务器 benchmark 工具，可以对条件 GET 进行吞吐量测试。使用 ab 工具模拟 100 个客户端进行 1000 次的请求来测试资源地址 `http://192.168.0.163:8080/simple-service10/webapi/rest`，测试报告如下所示。

```

//ab - Apache HTTP server benchmarking tool
//ab [ -c concurrency ] [ -H custom-header ] [ -n requests ] [http[s]://]
hostname[:port]/path
ab -n1000 -c100 http://192.168.0.163:8080/simple-service10/webapi/rest
Requests per second:      131.13 [#/sec] (mean)
Time per request:         762.574 [ms] (mean)
ab -n1000 -c100 -H 'If-Modified-Since:' http://192.168.0.163:8080/simple-
service10/webapi/rest/last_modified?userId=eric
Requests per second:      158.23 [#/sec] (mean)
Time per request:         632.006 [ms] (mean)

```

如上所示，-n 参数表示请求数，-c 参数表示并发客户端数。当没有使用缓存策略时，100 个客户端进行 1000 次请求得到的每秒请求次数是 131.13 次，每个请求的处理时间是 762.574 毫秒。同样的测试参数对支持条件 GET 的资源地址进行测试，得到的每秒请求次数是 158.23 次，每个请求的处理时间是 632.006 毫秒。

9.2 使用版本号优化服务

在 REST 服务的资源地址中加入版本号，可以方便地支持在 REST 服务接口升级后，对旧接口的长期支持。但是否应该在 REST 资源中使用版本号，REST 领域的声音并不一致。本节着眼于使用版本号为开发和运维人员带来效率上的提升的角度，讲述 REST 服务

中的版本号。

9.2.1 何时使用版本号

REST 的版本号 (version) 并不是必要存在的, 我们从 B/S 结构的网站和应用软件两个应用场景讨论 REST 服务中何时该使用版本号信息。

1. 网站

网站的特点是永远处于 Beta 版, 发版是开发团队和运维团队的日常工作。因此, 确保与外系统之间的 Web 服务始终有效是网站灰度测试不可或缺的环节。如果我们的 Web 服务升级, 必须在上线前确保外系统的接入没有问题。

最直接的办法就是提供带有版本号的地址, 在上线新版本时通知对方修改其 Web 服务接入地址。这个场景下, 外系统如果接入旧版本地址, 可以继续使用旧版本功能, 省去了因为我们升级导致的不兼容问题。此后, 外系统的开发人员可以视情况将其接口方法升级到新版本, 然后接入我们的新版地址。如果对方由于某种原因无法修改接入地址, 我们的系统应该保留旧的资源方法并提供兼容性解决方案。比如通过在访问者身份信息和请求头信息中额外定义的版本信息等方式, 相应地为其提供新版服务。

其实, REST 式的 Web 服务中使用版本号并不是必要的。如果必须使用, 笔者推荐将最新版本的资源地址和无版本号的资源地址一并使用, 这样做的好处是, 如果接入端不使用带有版本号的地址, 那么它接入的永远是最新版本的资源地址, 而无须因为服务端的频繁升级带来的无谓的修改。

2. 应用软件

如果我们开发的 REST 式的 Web 服务是企业管理平台一类的软件, 如果出现带版本号的资源地址, 那么该版本号应该和软件版本匹配, 退一步讲即使从来都没有被修改过, version 信息也应该相应地升级。其实, 在软件中使用版本号的必要性很低, 维护版本的价值本身就不大。笔者经历过一个对版本号依赖很强的产品, 该产品历经多年开发, 有多个小版本, 其中多数是为支持不同的客户开发的, 小版本之间缺乏共性, 因此 Web 服务中定义了版本号。这个例子中, 版本号的确有其存在的必要, 但排除历史原因, 一个优秀的设计是可以规避引入日后不必要的维护成本的。

9.2.2 如何使用版本号

黑格尔说: “存在即合理”。纵然笔者以浅薄的认识站在尽量不去使用版本号的行列, 但当需要使用版本号的需求摆在面前时, 头等大事还是如何在 REST 服务中实现 version。

1. 资源定义

REST 服务对版本号信息的支持有两种方式：通过资源地址（URL）或者 HTTP 头信息来实现。如下分别讲述两种实现。

(1) URL

通过资源地址实现的方式实现 REST 的版本信息，其 URL 形如下所示。

```
http://localhost:8080/simple-service10/webapi/rest/v1.0
http://localhost:8080/simple-service10/webapi/rest/v2.0
```

在服务的 URL 中，版本号信息作为资源路径的一部分加入在 context 与资源名称之间。上面定义了 1.0 版本和 2.0 版本。相应的资源方法的实现并不复杂，只需额外增加一个 @Path 注解，示例代码如下。

```
@GET
@Path("/v2.0")
@Produces(MediaType.TEXT_PLAIN)
public String getIt2() {
    ...
}
```

对应的测试代码分别测试了两个版本的资源地址的可用性，示例代码如下。

```
@Test
public void testVersion() {
    Response response = target("rest").path("v1.0").request().get();
    Link link = response.getLink("currentVersion");
    System.out.println(link);

    Response response2 = target("rest").path("v2.0").request().get();
    String result = response2.readEntity(String.class);
    System.out.println(result);
}
```

(2) HEAD

通过头信息支持版本号需要在 HTTP 头信息（HEAD）中指定一个特殊的属性，本示例使用 X-API-Version 作为版本信息的标识，示例代码如下。

```
@GET
@Path("/head-version")
@Produces(MediaType.TEXT_PLAIN)
public String getIt3(@Context final HttpHeaders headers) {
    String version = headers.getRequestHeaders().get("X-API-Version").get(0);
    if (version.equals("2"))
        return getIt2();
    ...
}
```

在这段代码中，资源方法 getIt3 首先从请求头信息中获取 X-API-Version 属性信息，该属性值代表请求的 REST 版本信息。

对应的测试代码通过定义请求头信息中的 X-API-Version 值，即可对不同版本的资源地

址进行访问，示例代码如下。

```
@Test
public void testHeadVersion() {
    Response response = target("rest").path("head-version").request()
        .header("X-API-Version", "2").get();
    String result = response.readEntity(String.class);
    System.out.println(result);
}
1 > GET http://localhost:9998/rest/head-version
1 > X-API-Version: 2
```

2. 过期定义

一个 REST 服务经历长期的开发后，也许需要对早期的 Web 服务资源地址进行去化。在正式废弃旧版本的资源地址之前，应该提供相当一段时间，比如间隔几个版本的时间的兼容性支持，并在 API 文档中给出最后支持时间，以便外系统或客户端有充分的升级准备。一种友好的通知版本变更的方式是可以通过响应头 Link，将新版本的资源地址响应给客户端，示例代码如下。

```
@GET
@Path("/v1.0")
@Produces(MediaType.TEXT_PLAIN)
public Response getIt1(@Context UriInfo uriInfo) {
    final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
    final URI uri = ub.replacePath("rest/v2.0").build();
    return Response.accepted().link(uri, "currentVersion").build();
}
```

在这段代码中，响应信息提供了 link 信息，其 rel 定义为 currentVersion 代表该资源的当前版本，uri 的值是当前版本的资源地址。

9.3 使用参数配置优化服务

Jersey 提供了丰富的配置参数，用于为 REST 服务特定的需求提供定制。本节将分 3 部分讲述 Jersey 的配置：通用部分、服务器端、客户端。

9.3.1 通用配置

Jersey 提供的通用配置类 org.glassfish.jersey.CommonProperties 位于 Jersey-Common 包内。在应用中通过修改通用配置的值，可以改变 Jersey 服务器端和客户端两侧的默认行为。比如，禁用不必要的自动探测功能，可以提高系统的整体性能。表 9-1 为 Jersey 通用配置参数参考。

表 9-1 Jersey 通用配置参数列表

| 常量名称 | 参数名称 | 说 明 |
|---------------------------------|---|--|
| FEATURE_AUTO_DISCOVERY_DISABLE | jersey.config.disableAutoDiscovery | 全局禁用自动探测特征。默认值 =false |
| JSON_PROCESSING_FEATURE_DISABLE | jersey.config.disableJsonProcessing | 禁用对 Json Processing (JSR-353) 特征的支持。默认值 =false |
| METAINF_SERVICES_LOOKUP_DISABLE | jersey.config.disableMetainfServices-Lookup | 禁用自动加载 META-INF/services 目录下的 SPI。默认值 =false |
| MOXY_JSON_FEATURE_DISABLE | jersey.config.disableMoxyJson | 禁用对 MOXYJson 特征的支持。默认值 =false |
| OUTBOUND_CONTENT_LENGTH_BUFFER | jersey.config.contentLength.buffer | 设置响应实体缓存的大小和响应头 Content-Length 的值。默认值 =8192 |

在 REST 应用中修改通用配置类 org.glassfish.jersey.CommonProperties 的值非常简单，示例代码如下。

```
public class AirApplication extends ResourceConfig {
    public AirApplication() {
        property(CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE,true);
        property(CommonProperties.JSON_PROCESSING_FEATURE_DISABLE,true);
        property(CommonProperties.METAINF_SERVICES_LOOKUP_DISABLE,true);
        property(CommonProperties.MOXY_JSON_FEATURE_DISABLE,true);
        property(CommonProperties.OUTBOUND_CONTENT_LENGTH_BUFFER,20480);
        packages("com.example.resource");
    }
}
```

在这段代码中，分别配置了禁用自动探测 feature、支持 JSR-353、自动加载 SPI 和支持 MOXY 的 JSON，并配置了响应实体缓存的大小为 20K。

同时，CommonProperties 类提供对服务端或者客户端的单边设定，相关代码如下。

```
public static final String FEATURE_AUTO_DISCOVERY_DISABLE = "jersey.config.disableAutoDiscovery";
public static final String FEATURE_AUTO_DISCOVERY_DISABLE_CLIENT = "jersey.config.client.disableAutoDiscovery";
public static final String FEATURE_AUTO_DISCOVERY_DISABLE_SERVER = "jersey.config.server.disableAutoDiscovery";
...
```

9.3.2 服务器端和客户端配置类

Jersey 提供的服务器端配置类 org.glassfish.jersey.server.ServerProperties 位于 Jersey-Server 包内。在应用中通过修改通用配置的值，可以改变 Jersey 服务器端的默认行为。如禁用不必要的自动探测功能，可以提高系统的整体性能。客户端配置类 org.glassfish.jersey.client.ClientProperties 位于 Jersey-Client 包内。在应用中通过修改通用配置的值，可以改变

Jersey 客户端的默认行为。如禁用不必要的自动探测功能，可以提高系统的整体性能。

9.4 Java 虚拟机调优

基于 JAX-RS 的项目，通常要关注服务器和开发环境的 Java 虚拟机配置。日常的关注点通常是内存溢出和内存泄漏。为了更好地解决 Java 虚拟机这个不是项目代码问题的问题，这里将从简述虚拟机开始。

9.4.1 虚拟机概述

在虚拟机加载类之前，Java 类的源代码首先被构建工具（如 Maven 等）通过调用编译器 javac 编译成 .class 字节码文件。虚拟机启动时，使用 ClassLoader 以双亲委派机制装载类的字节码文件，链接到内存，然后对类进行初始化（或者滞后初始化）。此时，Java 类就进入了虚拟机的内存，我们的关注点转到虚拟机的内存模型。虚拟机内存模型示意图如图 9-7 所示。

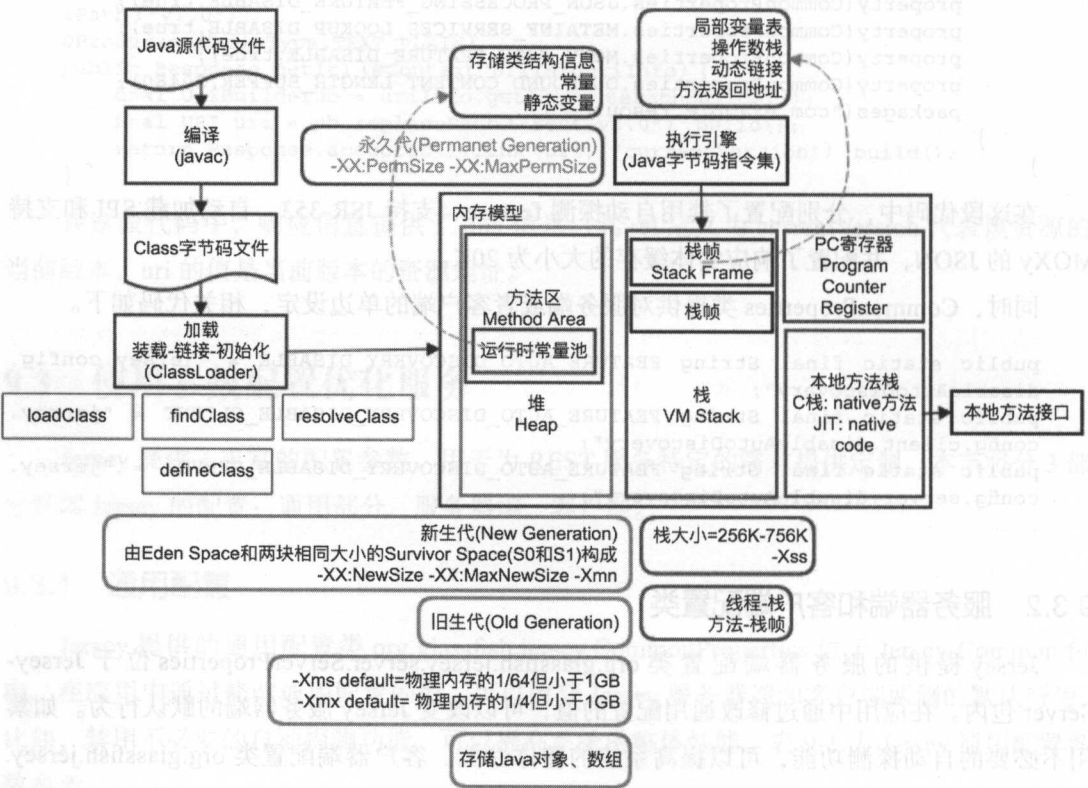


图 9-7 Java 虚拟机示意图

在图 9-7 中, Java 虚拟机大体上可分为堆和栈两部分。其中, 堆用于存储对象和数组, 堆内的资源是线程共享的。堆细分为新生代、旧生代和永久代, SUN 的虚拟机将永久代和方法区等同(永久代在 JDK8 中被驱除出 JVM)。方法区内还包括常量池和运行时常量池, 方法区用于存储类信息以及常量和静态变量等信息。垃圾回收器对堆和方法区的回收频率是不同的, 方法区内的资源通常不会被频繁地回收。非严格来说, 虚拟机栈、本地方法栈以及 PC 寄存器都可以视为栈。虚拟机栈与运行期的线程对应, 每启动一个线程就会在寄存器中注册一个指针, 指向内存中为其开辟的虚拟机栈。虚拟机栈内会为线程的方法开辟栈帧, 栈帧用于存储局部变量、操作数栈、方法返回值等信息。本地方法栈用于调用操作系统相关的本地方法接口, 本地方法是指使用 native 关键字修饰、使用 C 语言实现的方法, 因此本地方法栈又称 C 栈。本地方法栈同时存储 JIT 编译的方法, 这类方法由于在虚拟机栈中被频繁调用到一定次数, 被 JIT 编译成本地方法, 以提高性能。

可以使用 JDK 自带的工具 jmap 和 jinfo 来查看项目的虚拟机配置, 示例代码如下。

```
jmap -J-d64 -heap 6632
```

```
Heap Configuration:
MinHeapFreeRatio = 40
MaxHeapFreeRatio = 70
MaxHeapSize      = 1073741824 (1024.0MB)
NewSize          = 1310720 (1.25MB)
MaxNewSize       = 17592186044415 MB
OldSize          = 5439488 (5.1875MB)
NewRatio         = 2
SurvivorRatio    = 8
PermSize         = 21757952 (20.75MB)
MaxPermSize      = 268435456 (256.0MB)
G1HeapRegionSize = 0 (0.0MB)
```

在示例中, -J-d64 参数是 64 为 JDK 使用的参数, -heap 参数是打印堆信息, 最后的参数是 pid, 可以通过 jps 等方式获得。使用 jinfo 查看 JVM 的详细信息示例如下。

```
// 虚拟机参数值
jinfo -J-d64 -flags 6632
-Dosgi.requiredJavaVersion=1.5 -Xms1024m -Xmx1024m -XX:MaxPermSize=256m

// 堆初始值
jinfo -J-d64 -flag InitialHeapSize 6632
-XX:InitialHeapSize=1073741824

// 堆最大值
jinfo -J-d64 -flag MaxHeapSize 6632
-XX:MaxHeapSize=1073741824

// 新生代初始值
jinfo -J-d64 -flag NewSize 6632
-XX:NewSize=1310720
```

```
// 新生代最大值
jinfo -J-d64 -flag MaxNewSize 6632
-XX:MaxNewSize=18446744073709486080
```

```
// 永久代初始值
jinfo -J-d64 -flag PermSize 6632
-XX:PermSize=21757952
```

```
// 永久代最大值
jinfo -J-d64 -flag MaxPermSize 6632
-XX:MaxPermSize=268435456
```

在示例中，“-flags”打印显式定义的虚拟机参数值列表，“-flag 参数名”打印某项参数值。日常开发和调试中遇到的栈溢出可以调整虚拟机栈参数 -Xss，但如果虚拟机栈的大小不是根本问题，比如递归方法中不适当的编码产生死循环导致的栈溢出，还需要监控和跟踪到代码来发现和解决问题。

对于处理堆的异常或者错误，通常可归结为内存溢出和内存泄漏两个问题，浅析如下。

9.4.2 内存溢出与内存泄漏

内存溢出和内存泄漏的表象都是系统提供的内存空间不足以支撑应用的运行，但两者的实质是不同的。前者可以理解为进入虚拟机的资源总量比出去的资源总量多，占用内存的速率要快于垃圾回收的速率，这有可能是正常的业务逻辑，也有可能是设计上的缺陷；后者可以理解为进入虚拟机的某些资源存在引用并无法被垃圾回收，虚拟机失去了对这部分资源的回收能力，简述如下。

1. 内存溢出

内存溢出的原因是堆的大小无法承载内部的对象和数组了。这个问题简单地讲可以从两个方面解决。一是在代码中减少不必要的实例构造，这个途径会直接提高项目的性能。二是调整堆大小，-Xms 是设置堆的最小空间的参数，-Xmx 是最大空间参数。对于新生代空间的设置参数为 -XX:NewSize 和 -XX:MaxNewSize，当 -Xms 和 -Xmx 的值相同时，可以使用参数 -Xmn 来简化。对于永生代（SUN 虚拟机对应的是方法区）的设置，初始空间参数为 -XX:PermSize，最大空间参数为 -XX:MaxPermSize。注意，虚拟机一旦启动，堆的总大小就被固定，无法动态调整。

2. 内存泄漏

内存泄漏的原因是垃圾回收器无法对存在无效引用的对象进行回收导致的内存溢出。内存泄漏没有办法通过设置虚拟机参数来解决，因为导致这一问题的原因来自编码阶段，比如资源在使用完毕后没有被释放等。解决这一问题的本质是改进代码或者变更依赖包（项目

的依赖包并不总是让人放心的), 发现问题的途径不一而足, 通常是通过分析工具来排查泄漏点。常用的分析工具如表 9-2 所示。

表 9-2 Java Profiler 工具列表

| 工具名称 | 获取路径 |
|--|---|
| jvisualvm (Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool) | JAVA_HOME/bin |
| jmc (Oracle Java Mission Control) | JAVA_HOME/bin (JDK7u40+) |
| MAT (Memory Analyzer) | http://www.eclipse.org/mat |
| JProfiler | http://www.ej-technologies.com/products/jprofiler/overview.html |
| YourKit Profiler | http://www.yourkit.com |

分析过程包括实时监控和 dump 快照两种方式, 前一种通过工具本身提供的 GUI 界面即可实现, 后一种可以通过 JDK 自带的工具 jmap 来 dump 当前虚拟机快照, 命令示例如下:

```
jmap -dump:format=b,file=mem.bin<pid>
```

再做分析的机器执行, 然后将快照文件 (本例是 Ubuntu 系统下的 /var/local/my.mem.bin) 导入分析工具, 排查内存泄漏点, 命令示例如下。

```
sudo scp root@server:/home/erichan/mem.bin /var/local/my.mem.bin
```

分析过程包括实时监控和 dump 快照两种, 前一种通过工具本身提供的 GUI 界面即可实现, 后一种可以通过 JDK 自带的工具 jmap 来 dump 当前虚拟机快照, 然后将快照文件 (本例是 Ubuntu 系统下的 /var/local/my.mem.bin) 导入分析工具, 排查内存泄漏点。

在 Server 上执行 jmap -dump:format=b,file=mem.bin<pid>, 再做分析的机器执行 sudo scp root@server:/home/erichan/mem.bin /var/local/my.mem.bin。

另外, 也可以通过 psi-probe 这样的实时工具对运行期的服务器环境进行监控。如下给出 psi-probe 的安装示例。

下载 zip 格式的 psi-probe 工具包。

```
sudo wget https://psi-probe.googlecode.com/files/probe-2.3.3.zip -P /opt
```

将其中的 war 包部署到 tomcat 的应用路径下。

```
sudo unzip /opt/probe-2.3.3.zip -d /opt/  
cp /opt/probe.war /opt/apache-tomcat-7.0.42/webapps/
```

启动 tomcat, 然后访问该工具的 url, 这里 context 是 probe。通过地址 http://localhost:8080/probe/logs/index.htm 测试启动情况。

由于篇幅所限,虚拟机的部分就抛砖到此,读者可以找来专门讲述 Java 虚拟机的书来学习。值得称道的是,在这个领域国内已经出现几本品质优良的中文书籍了。

9.5 本章小结

本章讲述了使用 Java 进行 REST 开发中需要注意的性能和优化。9.1 节讲述了 HTTP 通信过程中的缓存机制和实践,9.2 节讲述了何时使用以及如何优化 REST API 的版本信息,9.3 节讲述了 Jersey 提供的参数配置,9.4 节简单介绍了 Java 虚拟机的情况、如何优化其配置以及基本的 JVM 问题解决办法。

9.4.2 内存溢出与内存泄漏

内存溢出 (Out of Memory) 是指 JVM 在运行过程中,由于堆内存不足,导致无法继续运行。内存泄漏 (Memory Leak) 是指 JVM 在运行过程中,由于某些对象无法被垃圾回收,导致堆内存占用量不断增加,最终导致内存溢出。

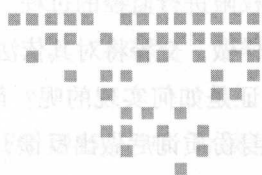
内存溢出和内存泄漏是 JVM 中常见的两种问题。内存溢出通常是由于堆内存不足导致的,而内存泄漏则通常是由于某些对象无法被垃圾回收导致的。内存溢出和内存泄漏都会导致 JVM 运行失败,因此需要及时发现并解决。

内存溢出的常见原因包括:堆内存不足、堆内存使用不当、堆内存使用过多等。内存泄漏的常见原因包括:静态变量、全局变量、未释放的资源等。

内存溢出的解决方法包括:增加堆内存、优化堆内存使用、使用垃圾回收器等。内存泄漏的解决方法包括:释放静态变量、全局变量、未释放的资源等。

2. 内存泄漏

内存泄漏是指 JVM 在运行过程中,由于某些对象无法被垃圾回收,导致堆内存占用量不断增加,最终导致内存溢出。内存泄漏通常是由于静态变量、全局变量、未释放的资源等导致的。



第 10 章

Chapter 10

REST 安全

REST 服务提供了统一接口和资源定位，简化了 Web 服务接口的设计和实现，降低了 Web 服务的复杂度。与此同时，易于识别和理解的 REST 接口也潜在着易于被攻击的危险。如果破坏者通过对 Web 服务资源地址的猜解，获取了删除某一资源的接口并对该 Web 服务进行攻击，很容易造成系统数据的破坏。因此，REST 式的 Web 服务的安全性至关重要。本章将对 REST 的安全性进行全面的讲述。

REST 的特点是无状态的，因此基本的安全手段就是应用系统实现 AAA 的过程，这个主题将是本章的核心。在讲述 AAA 之前，我想到了一位业内朋友的反馈：既然使用 REST 就是追求总体开发的简单，搞起 AAA 太麻烦。经过探讨我们总结了如下方案：在客户端和服务端同时使用时间令牌（或者类似的技术），每次请求使用令牌、用户名、密码、IP（这个元素对于内网中的多台主机不起作用）、请求 URL 等作为加密元素，服务器端接收请求后首先解密验证这些元素，再处理具体业务。安全信息可以考虑写入 HTTP HEAD。当然，这并不是一个完美的安全方案，但这个极简的实现比本章要讲述的更安全的证书认证的门槛低很多，比 HTTP 基本认证又安全很多。这里只想为读者抛砖，希望诸位因地制宜地采用更恰当的安全保护手段。书归正传，我们走进 AAA 的世界。AAA 是指认证（Authentication）、授权（Authorization）和计费（Accounting）。

什么是认证呢？认证就是系统识别访问者身份的过程。举个小例子，比如交警对司机进行交通检查，认证就是交警确认驾驶者是该车的准驾司机的过程。如果是双向认证，还包括驾驶者对交警身份确认的过程。什么是授权呢？还用这个小例子，授权就是当交警确认并记住（系统是通过 Cookie、加密的 URI 或者 Session）驾驶者的司机身份后，对该准驾类型

的司机授予的权限进行监控的过程。比如小型客车的司机在驾驶一辆大型货车，那么该驾驶者没有权限这样做，交警将对其依法处置（系统拒绝访问者访问，返回 HTTP 状态码 403）。

那么，认证是如何实现的呢？简单的认证方式是通过用户名一口令进行验证。访问者在服务器提出身份质询后做出反馈，即（加密）提交用户名和口令，服务器端接收该信息并对其进行验证。用户名一口令验证方式的实现比较简单、易行，但很容易被盗取和破解（即使经过加密处理）。

除了用户名一口令验证方式，还可以通过数字证书进行验证。其验证过程相对复杂。还以交通检查为例，过程如下。首先驾驶者出示驾照（数字证书，访问者身份的凭据）。交警拿到驾照后，首先核实驾照的真伪（数字证书是否是伪造的），判断的依据是核对驾照上是否有交警支队的印章（CA（Certificate Authority，权威的数字证书认证机构）签发的数字证书），还要判断这个印章的真伪（CA 是否在对方的信任证书列表中）。这个验证过程可能包括对 CA 身份的验证，比如交警支队的身份由交通部的印章来确认，直至追溯到双方都确信的 CA 出示的根证书。本例中，交警信任交警支队，所以交警支队的印章可以看作双方都信任的根证书）。接着，交警要查看证书的有效期等信息，保证证书合法、有效。到此，交警确认了驾照的合法性（服务器对客户端的数字证书本身的校验完毕）。

最后，交警会查看驾照上对驾驶者身份等信息的描述，来确认其身份和准驾车类型（这个步骤相当于服务器从证书中获取访问者的身份信息）。认证过程中，使用了诸多安全技术，比如加密算法、摘要算法、公钥/私钥对、证书生成—导入—导出—吊销等。授权的实现可以由容器级别的配置完成或者通过程序级别的注解和编码完成。

在了解了认证和授权的过程后，我们从简述理论到实践，逐一讲述 REST 中可以采用的认证和授权办法。

10.1 身份认证

HTTP 认证规范（RFC 2617）定义了两种 HTTP 身份认证方式：HTTP Basic（基本认证）和 HTTP Digest（摘要认证）。HTTP 认证是一种无状态的认证方式，服务器容器不会为这样的登录匹配 Session，身份信息随浏览器关闭而消失。Java 平台包含了 HTTP 的两种身份认证，此外还定义了 HTTP+HTML form-based authentication（表单认证）和证书认证。

基本认证、摘要认证和表单认证都是基于用户名一口令的认证机制，证书认证是基于证书的认证机制。基本认证和表单认证的请求过程需要提交用户的口令信息，而且对服务器的合法性缺乏判断依据，摘要认证和证书认证的请求过程将对服务器的合法性进行验证而且不直接提交用户的口令。

本节将对这 4 种认证方式逐一讲述。

10.1.1 基本认证

HTTP 的基本认证最初定义在 HTTP 1.0 规范 (RFC 1945) 中, 后续最新的定义包含于 HTTP 1.1 规范 (RFC 2616) 和 HTTP 认证规范 (RFC 2617)。HTTP 基本认证是指通过 Web 浏览器或其他客户端在发送请求时, 提供用户名和口令作为身份凭证的一种登录验证方式。在请求发送之前, 用户名和口令 (密码) 字符串通过一个冒号合并, 形如: Username:Password, 合并后的字符串经过 Base64 算法进行编码。

例如, 提供的用户名是 eric, 口令是 han, 合并后为 eric:han, 经过 Base64 算法进行编码后字符串为 ZXJpYzpoYW4=。

浏览器或客户端最终将经过 Base64 编码的字符串提交给服务器。Base64 编码的目的并不是实现安全与隐私, 而是为了将用户名和口令中的与 HTTP 协议不兼容的字符转换为兼容的字符集。

HTTP 基本认证的请求过程是一个质询 / 回应 (challenge/response) 的对话流程。即客户端请求一个需要身份认证的资源路径, 但是没有提供用户名和口令。服务器端响应 HTTP 状态码 401 (Unauthorized) 的应答, 并提供一个认证域 (Basic Realm)。客户端增加认证消息头, 内容为前述的 Base64 加密字符串, 形式为 Authorization: Basic base64encode (Username:Password), 并再次发起请求。服务器端接收请求并处理, 如果用户凭据非法或无效, 服务器可能再次返回 HTTP 状态码 401, 客户端可以再次提示用户输入口令。质询 / 回应会话流程是可以省略的, 即客户端第一次请求中就发送认证消息头。

10.1.2 摘要认证

摘要认证最初定义在 RFC 2069 (HTTP 的一个扩展: 摘要访问认证) 中, 以服务器生成的随机数来维护安全性。RFC 2069 中定义的认证响应由 HA1、HA2、A1 及 A2 组成。其中, nonce 是服务器端随机数。后续最新的定义包含于 HTTP 认证规范 (RFC 2617, 是 RFC 2069 的替代方案)。RFC 2617 引入了一系列安全增强的选项, 包括 QoP (Quality of Protection, 保护质量)、请求计数器 nc (nonceCount) 和 cnonce (clientNonce, 客户端随机数)。这些增强项有效地防止了通信过程中的明文攻击。RFC 2617 中对 A2、HA2 和 response 做了如下增强定义, 如表 10-1 所示。

HTTP 摘要认证同样遵循请求过程的质询 / 回应 (challenge/response) 对话流程。具体描述参考前一节的基本认证。

表 10-1 RFC 2069 和 RFC 2617 摘要认证参数列表

| 参数组合 | 说 明 |
|-----------------------------|---|
| A1=Username:Realm:Password | qop=auth/NONE 时, A2=method:digestURI |
| A2=method:digestURI | qop=auth-int 时, A2=method:digestURI:MD5(entityBody) |
| HA1=MD5(A1) | qop=NONE 时, response=MD5(HA1:nonce:HA2) |
| HA2=MD5(A2) | qop=auth/auth-int 时, response=MD5(HA1:nonce:nonceCount:clientNonce:qop:HA2) |
| response=MD5(HA1:nonce:HA2) | - |

10.1.3 表单认证

表单认证是一种基于 HTTP 协议，使用 HTML 的 Form 标签提交表单的认证形式。用户登录页面定义在 web.xml 文件的 form-login-page 字段中，在没有被认证前，访问者对资源地址的访问会被引导到该页面。访问者提交身份信息后，服务器接收并处理请求，如果认证通过将重定向到 welcome-file 字段定义的页面，如果失败将重定向到 form-error-page 字段定义的页面。相对于 HTTP 认证，表单认证允许开发者指定并设计具有良好用户体验的登录页面、登录成功页面和登录失败页面。

Java 平台处理表单认证的实现是在客户端页面（比如 HTML+Ajax、JSP、JSF 等）中，定义一个名称指定为 j_security_check 的请求方法，该方法传递两个名称固定的字段。用户名字段为 j_username，口令字段为 j_password，服务器端会根据上述固定名称对提交的表单进行解析和验证，示例如下。

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

10.1.4 证书认证

证书认证是通过数字证书认证身份的方式。这一方式从技术角度可以拆分为证书管理和通信协议两个环节。证书是静态的文件，为基于 SSL/TLS 协议的通信过程所用。

1. 证书

证书是包含公钥和密钥属主信息的文件。加密是为了将信息保密地、完整地传递出去，加密算法是实现这一过程的手段。持有加密算法的文件即为密钥，密钥分为公钥和私钥，在加密、解密环节，公钥用于加密，私钥用于解密。

当甲持有乙的证书后，甲就可以从该证书中获取乙的公钥。使用该公钥对数据进行加密，然后发送给乙，乙使用私有的私钥将数据解密。这一过程中，信息的保密性和完整性

得到了保障，但安全性中的端点认证无法通过加密解密解决。就是说，甲无法确认接收者就是乙。

证书认证解决了端点认证的两个问题。

第一个问题是验证接收者是乙。在签名环节，私钥用于签名，公钥用于校验。乙的公钥就像乙手写签名的快照一样，只有乙本人的签名（乙的私钥）才与之能匹配。第二个问题是验证证书的合法性。通信过程中，对方提交的证书说他是乙，那么甲何从相信证书及其所言呢？这需要使用或间接使用另一个权威证书来鉴定该证书的合法性，请读者回忆下前面说过的交警支队需要交通部来证明其真伪的例子，这就是证书链模型。该模型最简版本为甲乙互相信任（参考 6.3 节的第 5 例），这种情况下无需 CA 证书参与进来。最复杂的版本就是甲乙只相信最权威的 CA 认证机构，而他们持有的认证证书却都不是直接来自自己的机构签发的。通过证书链逐级上溯，最终，甲通过信任 CA 来认可当前证书是为乙颁发的，这样一来，只要乙的私钥签名在经过证书公钥校验后结果一致，甲就可以相信接收者就是乙。

证书的签发格式遵循 X.509 标准。X.509 是由国际电信联盟（ITU-T）制定的数字证书标准。证书的管理可以通过 OpenSSL（<http://www.openssl.org>）和 Java 平台的 keytool 工具来实现，本章实例部分将使用 keytool 来实现证书的管理。

2. SSL/TLS

SSL（Secure Socket Layer，安全套接层协议）是在传输通信协议（TCP/IP）上实现的一种安全协议，采用公钥技术保证数据的保密性和完整性。HTTPS 是运行在 SSL 协议之上的 HTTP，其本质还是 SSL 通信。IETF 组织（www.ietf.org）将 SSL 标准化后成为 TLS（Transport Layer Security，传输层安全协议）（RFC2246）。SSLv2 已经过时，TLS1.0 非常近似 SSLv3，本书余文使用 TLS 代替 SSL/TLS 的书写方式。

TLS 包含 3 个基本阶段：①对等协商密钥算法；②传输基于非对称密钥加密的数据和基于 PKI（public key infrastructure，公钥基础设施）证书的身份认证；③传输基于对称密钥加密的数据。

3. JSR219

在 Java 领域，证书管理对 TLS 的支持定义在 JSR219 中。JSR219 规范（Foundation Profile 1.1，Java 平台对安全领略支持的规范）包括 3 个标准：JCE（Java Cryptography Extension）、JSSE（Java Secure Sockets Extension）、JAAS（Java Authentication and Authorization Service）。

□ JCE 定义了 Java 对加密解密、密钥生成和协商以及 MAC（Message Authentication Code，消息认证码）算法的实现规范。

□ JSSE 定义了 Java 对安全传输和握手机制的实现规范，包含了对 SSL/TLS 的支持。

□ JAAS 是 Java 平台认证和授权的标准。JAAS 建立在 PAM (Pluggable Authentication Module, 可插入的认证模块) 安全体系结构之上, 一方面, 使业务逻辑和安全校验解耦; 另一方面, 提供了 23 种设计模式中责任链模式的校验过程, 使模块化的校验机制可插拔地无缝集成于业务平台中。

10.2 资源授权

通过前一节的讲述, 我们对 REST 应用中可以采用的 4 种身份认证方式有了认识。在有了用户的身份信息后, 服务器要做的事情就是识别该用户对 REST 应用中资源的访问权限。授权管理包括容器管理和应用管理。容器管理权限提供无编码的、可配置的、全局的管理方式, 零编码、方便统一管理。应用管理权限提供细粒度的权限管理, 通过编码使具体业务的权限分配更灵活。接下来我们分别讲述两种授权管理的实现。

10.2.1 容器管理权限

容器管理权限是指容器通过启动时从其配置文件中加载角色—权限信息, 在运行时对用户身份进行认证, 并对其角色的资源访问权限进行管控的方式。

1. 基于 Realm 的身份认证

Java 平台的 Servlet 容器和 Java EE 容器根据规范各自都提供了认证和授权的实现。本书以 Tomcat 为例展示容器管理权限功能。

在 Tomcat 中, 使用 Realm 作为用户身份识别的基本单位, Realm 这个概念类似 UNIX 下的 groups, Tomcat 在运行期根据其中定义的 role 与访问资源的匹配, 控制当前用户的访问权限。Realm 的配置通常在服务器配置文件 conf/server.xml 中, 定义格式。在 Tomcat 配置文件中对 Realm 的定义示例如下。

```
<Realm className="实现 org.apache.catalina.Realm 接口的类全名"... 该实现的其他属性.../>
```

Realm 项可以内置于任何一个容器项中, 这将影响该 realm 的作用域 (scope)。

□ 内置于 Engine 项: 该 Realm 将作用于所有 hosts 中的 web 应用, 除非在 Host 项或者 Context 项中被覆盖。

□ 内置于 Host 项: 该 Realm 将作用于所有 virtual host 中的 web 应用, 除非在 Context 项或者 Context 项中被覆盖。

□ 内置于 Context 项: 该 Realm 将只作用于该 web 应用。

Tomcat 提供了 6 个标准的 Realm 认证插件：JDBCRealm、DataSourceRealm、UserDatabaseRealm、JAASRealm、MemoryRealm（通过读取 XML 格式内存集合对象，获取认证信息。这是一个过时的 Realm，被 UserDatabaseRealm 取代）和 JNDIRealm（通过 JNDI 访问 LDAP 服务器，获取认证信息）。本章 10.3 节将通过 5 个示例来展示如何结合各种身份认证和 Tomcat 提供的 Realm 实现 REST 应用中的认证和授权。

2. 角色—权限配置

配置文件中定义的角色—权限信息是使用 XML 完成的。虽然在很多架构或者工具的发展过程中，已经将其 XML 格式的配置文件转为注解方式，但是权限配置使用 XML 定义的方式，层次更清晰。容器管理权限的配置讲述如下。

(1) 配置 security-constraint 元素

security-constraint 元素用来定义安全约束。包括可访问的资源、访问者身份和数据传输类型的约束。

1) web-resource-collection：web-resource-collection 元素用来标识需要限制访问的资源子集。在 web-resource-collection 元素中，可以定义 url-pattern（使用资源路径通配符定义的资源集合）和 http-method（HTTP 方法）。如果不存在 HTTP 方法，就将安全约束应用于所有的方法。

2) auth-constraint：auth-constraint 元素用于指定可以访问该资源集合的用户角色。如果没有指定 auth-constraint 元素，就将安全约束应用于所有角色。role-name 元素包含安全角色的名称。举例说明，在如下定义中 user 角色可以访问 /webapi/* 资源集合中所有的 GET 方法。

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>restful resources</web-resource-name>
    <url-pattern>/webapi/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

3) user-data-constraint：user-data-constraint 元素用来标识在客户端和 Web 容器之间传输数据的方式。transport-guarantee 元素必须具有如下的某个值：None，这意味着应用不需要传输保证；Integral，这意味着服务器和客户端之间的数据必须以某种方式发送，而且在传送中不能改变。Confidential，这意味着传输的数据必须是加密的数据。通常，TLS 使用

Integral 或 Confidential。

(2) 配置 login-config 元素

login-config 元素用来指定所使用的验证方法、域名和表单验证机制所需的特性。

1) auth-method : auth-method 指定验证方法。它的值为以下 4 种中的一个: Basic、Digest、Form 或 Client-cert。

2) realm-name: realm-name 指定验证中使用的域名。

3) form-login-config : form-login-config 指定基于表单认证的登录中使用的登录页面和登录失败页面。如果没有使用基于表单的验证,则忽略这些元素。form-login-page 用于指定显示登录页面的资源路径,form-error-page 则用于指定用户登录失败时显示出错页面的资源路径。

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

(3) security-role 元素

security-role 元素指定用于安全约束中的安全角色的声明。role-name 指定角色名称。

```
<security-role>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <role-name>user</role-name>
</security-role>
```

10.2.2 应用管理权限

相对于容器管理权限,应用管理权限是指容器中的每个应用各自通过编码和注解完成对资源路径访问的权限管理。

1. JSR 250

JSR 250 (Common Annotations for the Java Platform 1.1, Java 平台通用注解标准)提供了安全访问注解,包名为 javax.annotation.security。应用代码通过使用安全访问注解对资源路径进行定义,实现权限管理。

注解 @PermitAll、@DenyAll 和 @RolesAllowed 用来定义方法级别的访问权限。使用 @DenyAll 注解定义的方法不允许任何角色访问。使用 @PermitAll 注解定义的方法允许所有角色访问。使用 @RolesAllowed 注解定义的方法允许指定的角色访问。如果注解在类级

别，适用于类中所有的方法。如果注解在方法级别，则方法级别的权限注解优先于覆盖类级别的权限注解。

注解 `@DeclareRoles` 和 `@RunAs` 用来定义类级别的访问权限。注解 `@RunAs` 注解定义的方法允许执行的角色必须存在于安全领域中，并映射到用户或组。`@DeclareRoles` 注解用来定义允许执行该类的安全角色。`@DeclareRoles("admin")` 相当于如下的 XML 格式的定义。

```
<security-role>
  <role-name>admin</role-name>
</security-role>
```

2. JAX-RS2

JAX-RS2 规范地定义了一个安全上下文接口 `Security Context`，通过该接口的实现可以在运行时获取当前用户的身份信息，从而通过编码方式，动态地为当前用户提供资源访问的权限。

Jersey2.x 提供了 `DynamicFeature` 接口的实现类 `RolesAllowedDynamicFeature`，REST 服务或应用通过注册该类即可使用 JSR 250 的注解和 JAX-RS2 的 `SecurityContext` 动态管理资源访问权限。其工作原理是 REST 应用使用注册的 `RolesAllowedDynamicFeature` 从 `SecurityContext.isUserInRole()` 获取用户角色并和类以及方法中使用的 JSR 250 注解匹配，如果失败即返回没有权限的错误（HTTP 状态码 403）给客户端。

10.3 认证与授权实现

前面两节分别讲述了认证和授权的实现方式，本节将结合不同的认证和授权方式，为读者呈现 5 种实现。

阅读指南

10.3 节的示例源代码地址为：<https://github.com/feuyeux/jax-rs2-guide-II/tree/master/>

10.3.security-rest。

10.3.1 基本认证与 JDBCRealm

本示例使用 HTTP 基本认证结合 Tomcat 提供的 `JDBCRealm` 实现认证与授权。本例所示的场景为以 `webapi/` 开头的资源路径需要用户登录访问。其中获取图书列表的只读资源路径 `http://localhost:8080/security-rest/webapi/books` 可以被 `admin` 角色和 `user` 角色访问，其他

以 webapi/ 开头的资源路径只有 admin 角色可以访问。

阅读指南

Realm 的概念在 10.2.1 节已经介绍。有关 Tomcat 的 Realm 详情请参考官方文档，地址是：<http://tomcat.apache.org/tomcat-8.0-doc/realm-howto.html>。

1. 创建 Realm 所需的数据表

JDBCRealm 通过 JDBC 访问关系型数据库获取认证信息，首先我们从数据库脚本入手，逐步完成认证与授权的实现。本例使用的 MYSQL 数据库，脚本信息如下。

```
// 导出数据库脚本的命令为: mysqldump simpleservicebook -uroot -p security.sql
// 导入数据库脚本的命令为: mysql -uroot -p < security.sql
DROP DATABASE IF EXISTS "simple_service_book";
CREATE DATABASE "simple_service_book";
USE "simple_service_book";
CREATE TABLE "simple_book" (
    "BOOKID" int(11) NOT NULL AUTO_INCREMENT,
    "BOOKNAME" varchar(128) DEFAULT NULL,
    "PUBLISHER" varchar(128) DEFAULT NULL,
    PRIMARY KEY ("BOOKID"),
    UNIQUE KEY "BOOKID" ("BOOKID")
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES "simple_book" WRITE;
INSERT INTO "simple_book" VALUES (1, 'Java Restful Web Service 使用指南', 'cmpbook'),
(2, 'JSF2 和 RichFaces4 使用指南', 'phei');
UNLOCK TABLES;
CREATE TABLE "user_roles" (
    "user_name" varchar(15) NOT NULL,
    "role_name" varchar(15) NOT NULL,
    PRIMARY KEY ("user_name", "role_name")
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES "user_roles" WRITE;
INSERT INTO "user_roles" VALUES ('caroline', 'user'), ('eric', 'admin');
UNLOCK TABLES;
CREATE TABLE "users" (
    "user_name" varchar(15) NOT NULL,
    "user_pass" varchar(15) NOT NULL,
    PRIMARY KEY ("user_name")
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES "users" WRITE;
INSERT INTO "users" VALUES ('caroline', 'zhang'), ('eric', 'han');
UNLOCK TABLES;
```

这段脚本创建了名为 simple_service_book 的数据库，该数据库包含名为 simple_book 的表用来展示 REST 请求资源，名为 user_roles 的表和名为 users 的表用来展示 JDBCRealm 认证和授权。本例包含两个用户：一个是角色为 admin 的 eric 用户，一个是角色为 user 的

caroline 用户。认证通过后，两个用户具有的身份不同，将会有不同的权限。

2. 配置 JDBCRealm

根据 Tomcat 官方网站提供的文档 <http://tomcat.apache.org/tomcat-7.0-doc/config/realms.html>，服务器配置文件 \$CATALINA_BASE/conf/server.xml 设置如下所示。

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
driverName="org.gjt.mm.mysql.Driver"
connectionURL="jdbc:mysql://localhost:3306/simple_service_book?user=root&password=root"
userTable="users"
userNameCol="user_name"
userCredCol="user_pass"
userRoleTable="user_roles"
roleNameCol="role_name"/>
```

在这段脚本中，用户表为 users，用户名字段为 user_name，口令字段为 user_pass；角色表为 user_roles，角色名字段为 role_name。

\$CATALINA_BASE 路径可以是 Tomcat 的物理路径，也可以是 IDE 集成环境中的相对路径。更灵活的设置方式是使用后者，即在 IDE 集成环境中对服务器配置，以方便切换不同配置和断点调试。以 Eclipse 为例，在 Servers 实例的内置 Tomcat 中，配置 server.xml 文件，如图 10-1 所示。

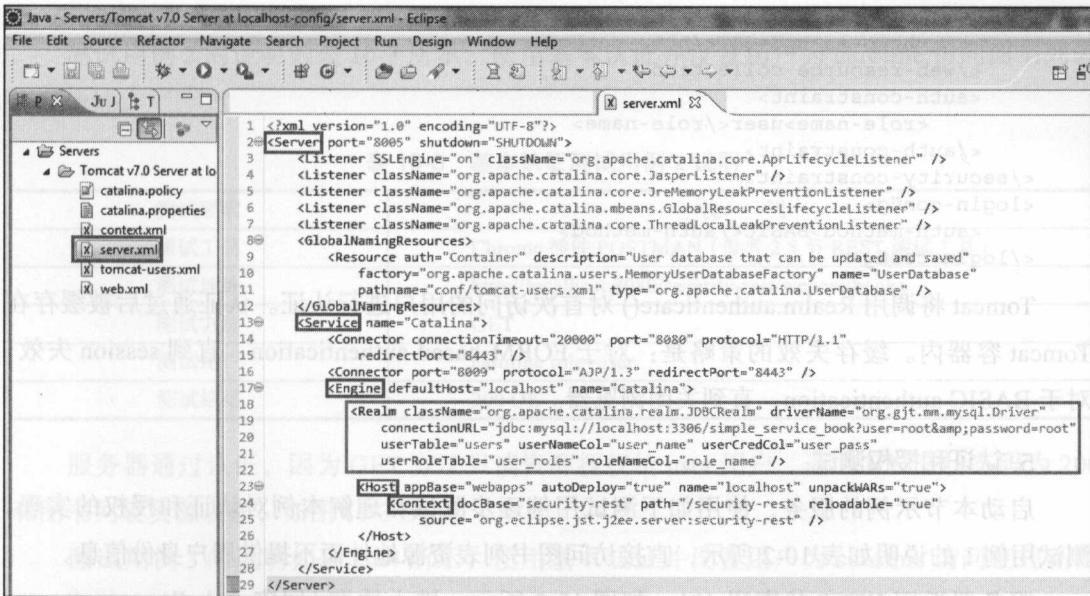


图 10-1 Eclipse 中定义 Tomcat 配置示意图

3. 数据库驱动

复制 MySQL 的 JDBC 驱动到 \$CATALINA_HOME/lib 目录。使用 Maven 的项目可以从本地仓库取得, 否则网上搜吧。本地仓库地址为 M2_REPO/mysql/mysql-connector-java/5.1.25/mysql-connector-java-5.1.25.jar (仓库地址路径示例: M2_REPO=C:\Users\hanl.m2\repository)。

4. 配置应用

Web 配置文件 /security-rest/src/main/webapp/WEB-INF/web.xml 摘要如下, 逻辑如前所述。

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/webapi/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>UPDATE</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/webapi/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

Tomcat 将调用 Realm.authenticate() 对首次访问的用户进行认证。认证通过后被缓存在 Tomcat 容器内。缓存失效的策略是: 对于 FORM-based authentication, 直到 session 失效; 对于 BASIC authentication, 直到关闭浏览器。

5. 认证和授权测试

启动本节示例的服务, 使用如下测试用例逐步测试和理解本例对认证和授权的实现。测试用例 1 的说明如表 10-2 所示, 直接访问图书列表资源地址而不提供用户身份信息。

服务器返回 HTTP 状态码 401, 如图 10-2 所示, 进入质询/回应 (challenge/response) 的对话流程, 流程参见后续测试用例。

表 10-2 基本认证测试用例 1 说明

| 测试要素 | 说 明 |
|------|--|
| 测试工具 | Chrome 插件 POSTMAN (参考 2.5 节 REST 调试工具) |
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | GET |
| 测试用户 | 无 |
| 测试结果 | 401 Unauthorized |



图 10-2 基本认证测试用例 1 结果

测试用例 2 的说明如表 10-3 所示，使用基本认证并设置用户为 caroline 访问图书列表资源地址。

表 10-3 基本认证测试用例 2 说明

| 测试要素 | 说 明 |
|------|--|
| 测试工具 | Chrome 插件 POSTMAN (参考 2.5 节 REST 调试工具) |
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | GET |
| 测试用户 | caroline role=user |
| 测试结果 | 200 OK |

服务器通过认证，因为 GET 方法只读资源授权给 user 用户，返回 HTTP 状态码为 200 和图书列表资源表述，如图 10-3 所示。

测试用例 3 的说明如表 10-4 所示，使用基本认证并设置用户为 caroline 访问创建新图书的资源地址。

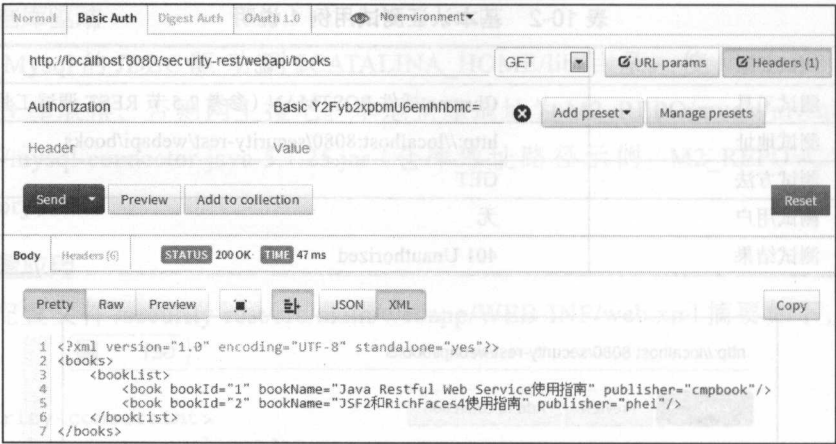


图 10-3 基本认证测试用例 2 结果

表 10-4 基本认证测试用例 3 说明

| 测试要素 | 说 明 |
|------|--|
| 测试工具 | Chrome 插件 POSTMAN (参考 2.5 节 REST 调试工具) |
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | POST |
| 测试用户 | caroline role=user |
| 测试结果 | 403 Forbidden |

服务器通过认证，因为创建新图书的资源地址只授权给 admin 用户，返回 HTTP 状态码为 403 请求被禁止，如图 10-4 所示。

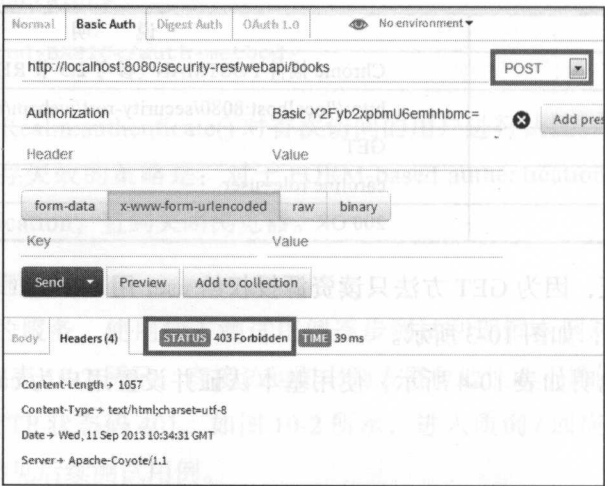


图 10-4 基本认证测试用例 3 结果

测试用例 4 的说明如表 10-5 所示，使用基本认证并设置用户为 eric 访问创建新图书的资源地址。

表 10-5 基本认证测试用例 4 说明

| 测试要素 | 说 明 |
|------|--|
| 测试工具 | Chrome 插件 POSTMAN (参考 2.5 节 REST 调试工具) |
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | POST |
| 测试用户 | eric role=admin |
| 测试结果 | 200 OK |

服务器通过认证，因为创建新图书的资源地址授权给 admin 用户，返回 HTTP 状态码为 200 和创建完毕的新图书资源表述，如图 10-5 所示。

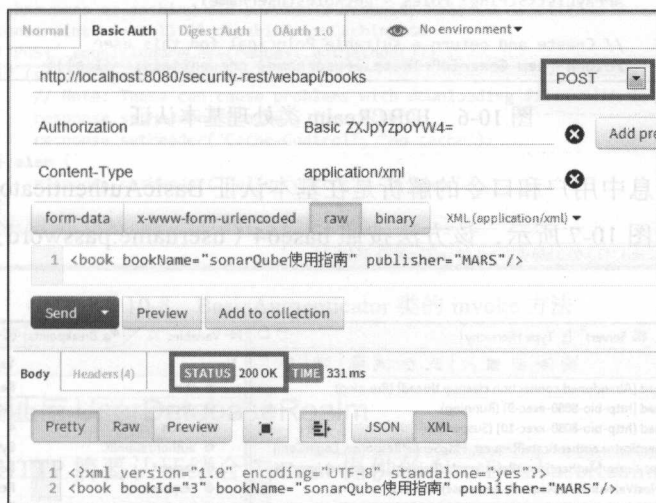


图 10-5 基本认证测试用例 4 结果

到此，我们通过 4 个测试用例对示例的场景进行了覆盖性的功能测试，本示例如预期完成了既定功能。最后，通过断点调试了解一下 Tomcat 如何使用 JDBCRealm 完成 HTTP 基本认证功能的。

6. BASIC 认证

阅读指南

本例使用的 Tomcat 版本为 7.0.42，Maven 仓库源代码地址为：C:\Users\hanl.m2\repository\org\apache\tomcat\tomcat-catalina\7.0.42\tomcat-catalina-7.0.42-sources.jar。

以 Debug 的方式在 IDE 中启动本示例，并使用上述的“测试用例 4”向服务器发出请求。如图 10-6 所示，通过断点的栈信息可以看到，JDBCRealm 类的 authenticate 方法最终处理客户端请求信息和数据库身份信息的匹配。

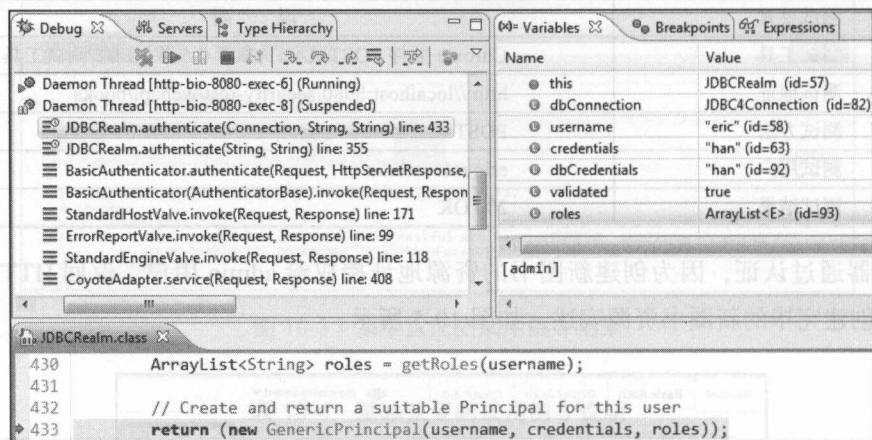


图 10-6 JDBCRealm 类处理基本认证

客户端请求信息中用户和口令的解析是在基本认证 BasicAuthenticator 类的 authenticate 方法中完成的，如图 10-7 所示，该方法按照 base64 (username:password) 的格式对字符串进行了解码。

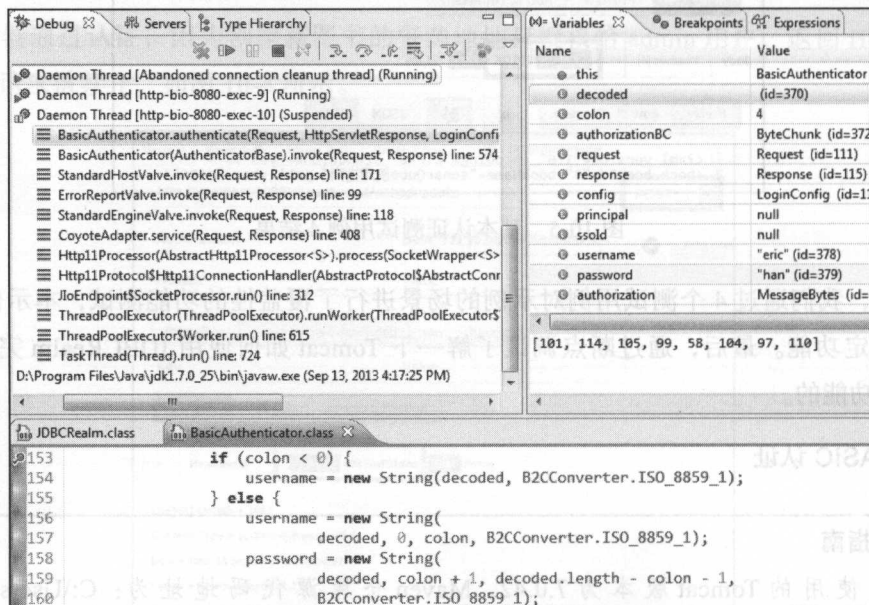


图 10-7 BasicAuthenticator 类的 authenticate 方法

最后, 如图 10-8 所示, BasicAuthenticator 类的 invoke 方法对响应头进行了修改。

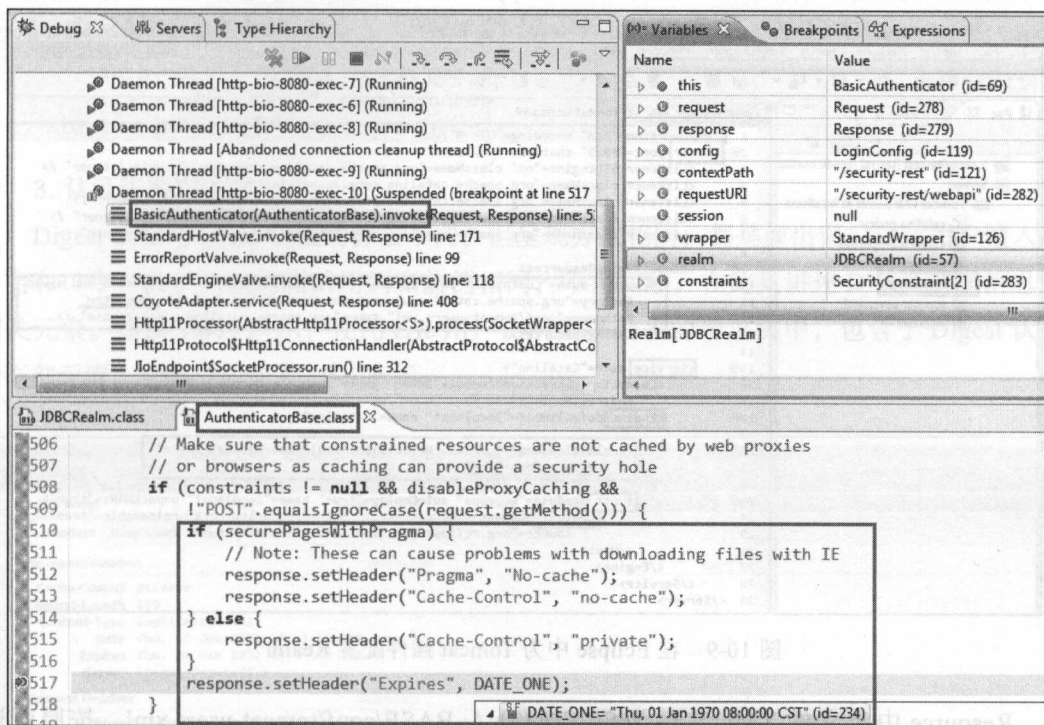


图 10-8 BasicAuthenticator 类的 invoke 方法

10.3.2 摘要认证与 UserDatabaseRealm

本示例使用 HTTP 摘要认证结合 Tomcat 提供的 UserDatabaseRealm 实现认证与授权。为了简化示例逻辑, 本例所示的授权场景和前例相同。不同的是 UserDatabaseRealm 通过读取 XML 格式的 JNDI 资源 (默认使用 conf/tomcat-users.xml) 获取认证信息。因此本例不需要数据库的支持。

1. 配置 UserDatabaseRealm

同前例, 服务器配置文件 \$CATALINA_BASE/conf/server.xml 添加 Realm, 其资源通过名称匹配在 Resource 中定义的 XML 文件。

```

<Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="tom" />
<Resource auth="Container" description="User database that can be updated and saved" factory="org.apache.catalina.users.MemoryUserDatabaseFactory" name="tom" pathName="conf/tomcat-users.xml" type="org.apache.catalina.UserDatabase" />

```

同前例，Eclipse 内置的 Tomcat 配置如图 10-9 所示。

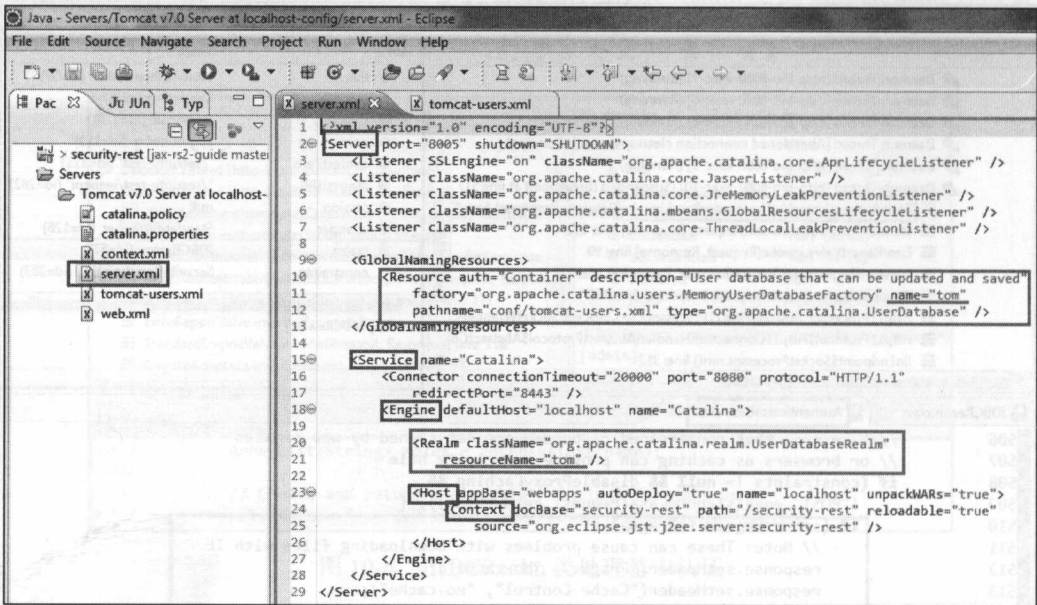


图 10-9 在 Eclipse 中为 Tomcat 插件配置 Realm

Resource 中定义的 XML 文件为 \$CATALINA_BASE/conf/tomcat-users.xml，按照前述的逻辑，定义如下。

```

<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
  <role rolename="admin" />
  <role rolename="user" />
  <user name="eric" password="han" roles="admin" />
  <user name="caroline" password="zhang" roles="user" />
</tomcat-users>

```

Eclipse 内置 Tomcat tomcat-users 配置如图 10-10 所示。

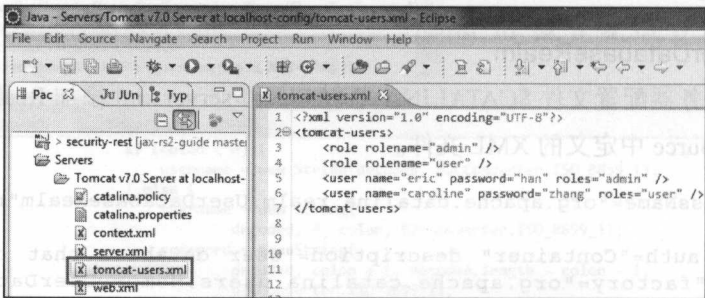


图 10-10 在 Eclipse 中为 Tomcat 插件配置用户权限

2. 配置应用

同前例，配置 /security-rest/src/main/webapp/WEB-INF/web.xml 文件。此处省略相同部分。

```
<login-config>
  <auth-method>DIGEST</auth-method>
  <realm-name>drealm</realm-name>
</login-config>
```

3. 认证和授权测试

Digest 认证与 Basic 认证的算法不同，但呈现方式相同，都是弹出窗口，让用户输入用户名和口令。由于本例的测试用例和前例相同，不再冗述，在此只讲述 Digest 请求信息的相关元素。如图 10-11 所示，使用 eric 用户访问图书列表的请求头中，包含了 Digest 认证的全部元素。详细分析如表 10-6 所示。

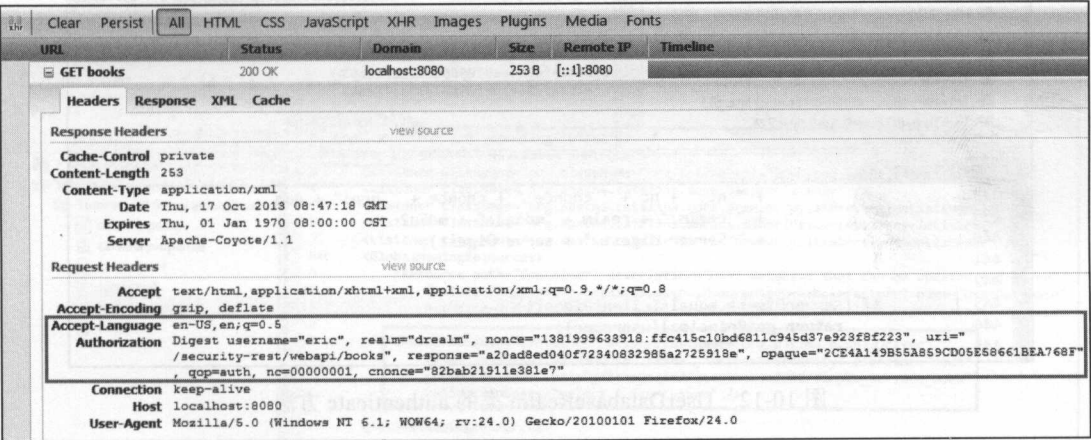


图 10-11 摘要认证头信息

表 10-6 摘要认证参数列表

| 认证条目 | 简 述 |
|--|-------------------------|
| username="eric" | 用户名 |
| realm="drealm" | Realm 名称 |
| nonce="1381999633918:ffc415c10bd6811c945d37e923f8f223" | 服务器端随机数 |
| uri="/security-rest/webapi/books" | REST 服务资源地址 |
| response="a20ad8ed040f72340832985a2725918e" | MD5 (HA1:nonce:HA2) 计算值 |
| opaque="2CE4A149B55A859CD05E58661BEA768F" | 服务器端质询响应信息 |
| qop=auth | 保护质量 |
| nc=00000001 | 客户端请求计数器 |
| cnonce="82bab21911e381e7" | 客户端随机数 |

4. Digest 认证

图 10-12 所示, 通过断点栈可以观察到, `UserDatabaseRealm` 类的 `authenticate` 方法完成了摘要认证的算法, 具体为 `md5 (md5 (username:realm:password) :nonce:nc:cnonce:qop:md5 (httpmethod:uri))`。请参考摘要认证类 `DigestAuthenticator`。

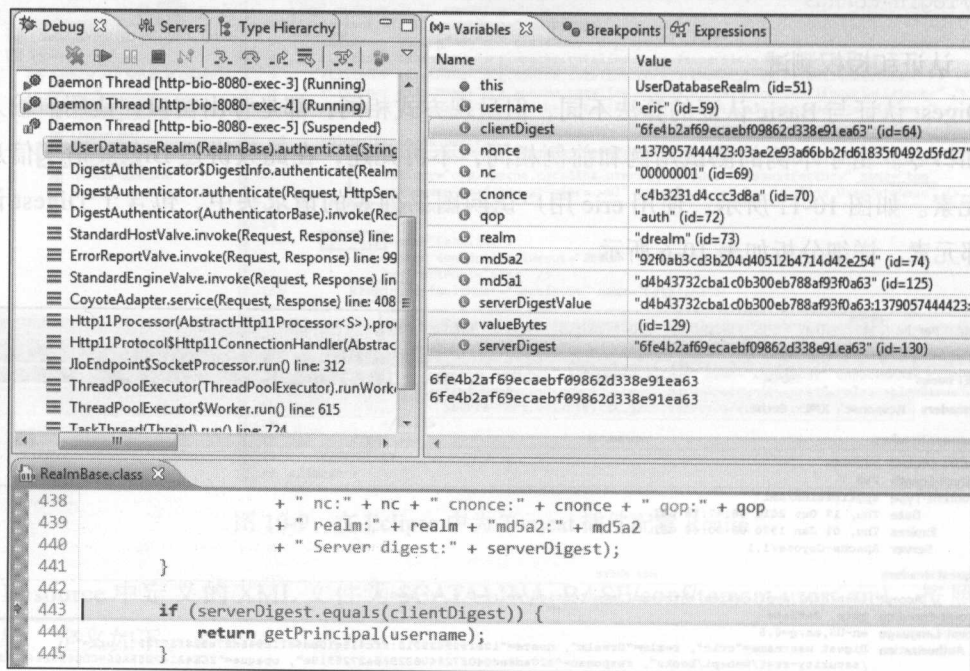


图 10-12 `UserDatabaseRealm` 类的 `authenticate` 方法

10.3.3 表单认证与 `DataSourceRealm`

本示例使用 FORM 认证结合 Tomcat 提供的 `DataSourceRealm` 类实现认证与授权。为了简化示例逻辑, 本例所示的授权场景和前例相同。`DataSourceRealm` 通过 JNDI 访问关系型数据库, 获取认证信息。`DataSourceRealm` 与相比, 数据库连接使用容器提供的全局数据源 (Data Source), 而不是在 Realm 中自行配置。我们首先需要创建 Realm 所需的数据表, 同 6.3.1 节的数据库脚本, 使用 `mysql -uroot -p < security.sql` 命令导入。

1. 配置 `DataSourceRealm`

同前例, 服务器配置文件 `$CATALINA_BASE/conf/server.xml` 添加 Realm, 首先配置容器的全局数据源。

```
<Resource auth="Container"
```



```

driverClassName="org.gjt.mm.mysql.Driver"
type="javax.sql.DataSource"
name="jdbc/rest-security"
username="root" password="root"
url="jdbc:mysql://localhost:3306/simple_service_book"
validationQuery="select 1 from users"/>

```

然后设置 DataSourceRealm 并使用该数据源。

```

<Realm className="org.apache.catalina.realm.DataSourceRealm"
dataSourceName="jdbc/rest-security"
userTable="users"
userNameCol="user_name"
userCredCol="user_pass"
userRoleTable="user_roles"
roleNameCol="role_name"/>

```

同 10.3.1 节示例，Eclipse 内置的 Tomcat 配置如图 10-13 所示。

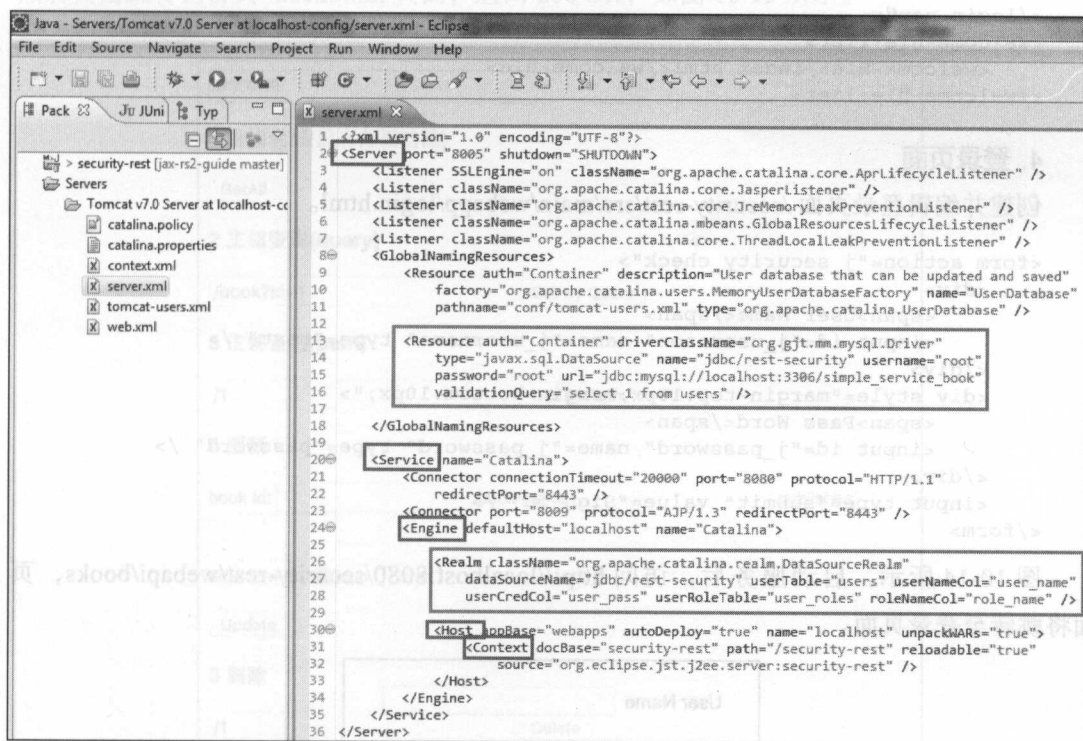


图 10-13 在 Eclipse 中为 Tomcat 插件配置 Realm

2. 数据库驱动

同 10.3.1 节示例，复制 Mysql 的 JDBC 驱动到 \$CATALINA_HOME/lib 目录。

3. 配置应用

同 10.3.1 节示例，配置 /security-rest/src/main/webapp/WEB-INF/web.xml 文件。此处省略相同部分。

```
<resource-ref>
  <description>MySQL DB Connection Pool</description>
  <res-ref-name>jdbc/rest-security</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
<welcome-file-list>
  <welcome-file>/index.html</welcome-file>
</welcome-file-list>
```

4. 登录页面

创建并编辑登录页面 /security-rest/src/main/webapp/login.html。

```
<form action="j_security_check">
  <div>
    <span>User Name</span>
    <input id="j_username" name="j_username" type="text" />
  </div>
  <div style="margin-top:10px;margin-bottom:10px;">
    <span>Pass Word</span>
    <input id="j_password" name="j_password" type="password" />
  </div>
  <input type="submit" value="Sign In" />
</form>
```

图 10-14 所示，启动服务后，访问 <http://localhost:8080/security-rest/webapi/books>，页面将跳转至登录页面。

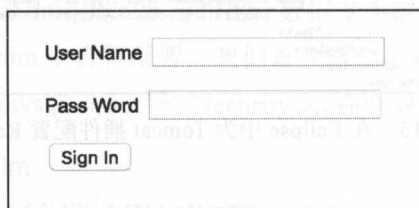


图 10-14 登录页面

如下是详细的测试用例。

5. 认证和授权测试

测试用例 1 (见表 10-7), 访问地址 `http://localhost:8080/security-rest/`, 页面跳转至登录页面。

表 10-7 摘要认证测试用例 1

| 测试要素 | 说 明 |
|------|---|
| 测试地址 | <code>http://localhost:8080/security-rest/</code> |
| 测试方法 | FORM jsecuritycheck |
| 测试用户 | <code>eric role=admin</code> |
| 测试结果 | 302 Found |

当用户信息提交后, 服务器处理登录信息返回 HTTP 状态码 302 (Found), 重定向请求, 然后页面跳转到首页 `index.html` 页面, 返回 200 OK, 如图 10-15 所示。

测试结果

1 查询全部(没有分页功能)

GetAll

2 主键查询(query)

/book?id=1

Get By Query

3 主键查询(path)

/1

Get By Path

4 更新

book Id:

book name:

publisher:

data format: ☒ json ☐ xml

Update

5 删除

/1

Delete

6 新增

book name:

publisher:

data format: ☒ json ☐ xml

Create

图 10-15 摘要认证测试用例 1 结果

测试用例 2，使用 admin 角色登录后，测试是否具有访问创建新图书的资源地址的权限。返回 HTTP 状态码 200 OK，如表 10-8 所示。

表 10-8 摘要认证测试用例 2

| 测试要素 | 说 明 |
|------|--|
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | POST |
| 测试用户 | eric role=admin |
| 测试结果 | 200 OK |

测试用例 3，使用 user 角色登录后，测试是否具有访问创建新图书的资源地址的权限。返回 HTTP 状态码 403 Forbidden，如表 10-9 所示。

表 10-9 摘要认证测试用例 3

| 测试要素 | 说 明 |
|------|--|
| 测试地址 | http://localhost:8080/security-rest/webapi/books |
| 测试方法 | POST |
| 测试用户 | caroline role=user |
| 测试结果 | 403 Forbidden |

6. FORM 认证

图 10-16 所示，通过断点栈可以观察到，DataSourceRealm 类的 authenticate 方法被表单认证类 FormAuthenticator 类的 authenticate 调用，匹配请求信息和数据库认证信息。

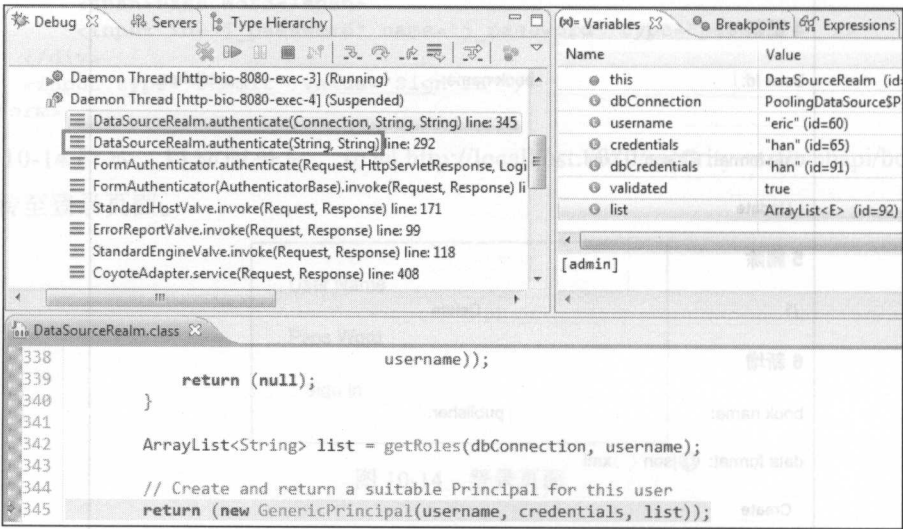


图 10-16 DataSourceRealm 类的 authenticate 方法

10.3.4 Form 认证和 JAASRealm

JAAS (Java Authentication & Authorization Service, Java 认证和授权服务) (JSR196 标准) 是 Java 平台认证和授权的标准规范, JAAS 只能使用 Form 认证方式获取用户登录信息。Tomcat 提供了基于 JAAS 的 JAASRealm, 本示例将使用 Form 认证和 JAASRealm 展示一个 JAAS 认证授权的实现过程。

1. 创建 Realm 所需的数据表

同 6.3.1 节的数据库脚本, 使用 `mysql -uroot -p < security.sql` 命令导入。

2. 配置 JAASRealm

配置 `$CATALINA_BASE/conf/server.xml`, 添加 JAASRealm 的定义。

```
<Context docBase="security-rest" ...>
<Realm className="org.apache.catalina.realm.JAASRealm"
      appName="RestJaasRealm"
      roleClassNames="com.example.jaas.RestRolePrincipal"
      userClassNames="com.example.jaas.RestUserPrincipal"/>
</Context>
```

Realm 定义在 context 中, 否则会导致角色 POJO 字段 `roleClassNames` 和用户 POJO 字段 `userClassNames` 中定义的类找不到。appName 定义的名字和 JAAS 配置文件 `restJaas.conf` 中定义的须一致。

Eclipse 内置 Tomcat 配置, 如图 10-17 所示。

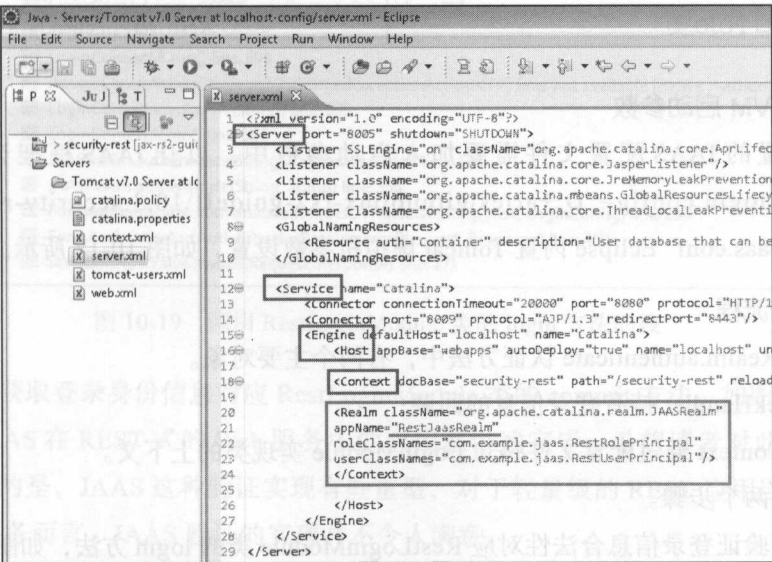


图 10-17 在 Eclipse 中为 Tomcat 插件配置 JAASRealm

3. JAAS 配置文件

实现 JAAS 需要为 JAAS 框架提供一个配置文件，本例使用 /security-rest/src/main/resources/restJaas.conf。RestJaasRealm{ com.example.jaas.RestLoginModule required; }; 名称 RestJaasRealm 与图 10-17 中的 appName 须一致。第一个参数是登录模块的类全名；第二个参数的取值见表 10-10 所示。

表 10-10 JAAS 配置行第二个参数取值和含义

| 参数值 | 说 明 |
|------------|----------------------------------|
| Required | 该模块必须认证用户，如果认证失败，使用其他登录模块认证 |
| Requisite | 如果认证失败，将终止认证 |
| Sufficient | 如果认证成功，即获得登录认证；如果认证失败，使用其他登录模块认证 |
| Optional | 认证将继续下去，即使该模块认证成功 |

4. JAAS 实现类

JAAS 的实现类目录为 10.3.security-rest\src\main\java\com\example\jaas，其内定义了 4 个 JAAS 实现类，如表 10-11 所示。

表 10-11 JAAS 实现类列表

| 实现类作用 | 实现类文件名 |
|------------------------|------------------------|
| LoginModule 实现类 | RestLoginModule.java |
| LoginModule 实现类的数据库操作类 | RestLoginDao.java |
| Role 接口 POJO 类 | RestRolePrincipal.java |
| User 接口 POJO 类 | RestUserPrincipal.java |

5. 配置 JVM 启动参数

前面设置的 JAAS 配置文件需要加入启动参数中，以让 JAAS 框架识别。-Djava.security.auth.login.config="D:+aries\github\jax-rs2-guideII\10.3.security-rest\src\main\resources\restJaas.conf" Eclipse 内置 Tomcat 虚拟机参数设置，如图 10-18 所示。

6. JAAS 流程

在 JAASRealm.authenticate 认证方法中，有两个主要对象。

- ❑ CallbackHandler 持有登录信息的回调；
- ❑ LoginContext 通过配置文件感知 LoginModule 实现类的上下文。

认证分为两个步骤。

第一步，验证登录信息合法性对应 RestLoginModule 类的 login 方法，如图 10-19 所示。

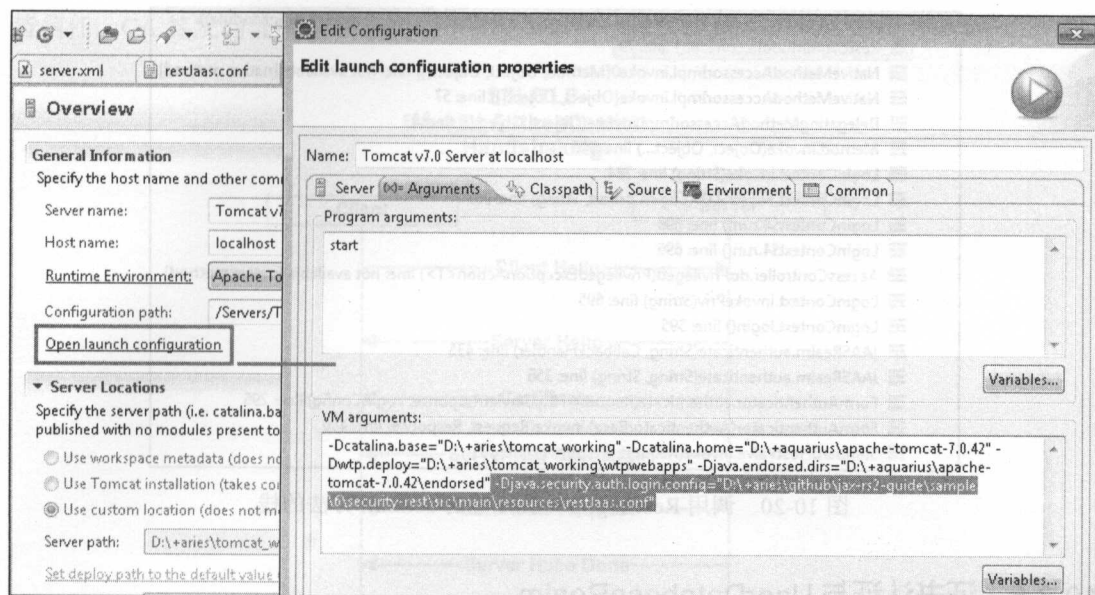


图 10-18 在 Eclipse 中为 Tomcat 插件配置支持 JAAS 的启动参数

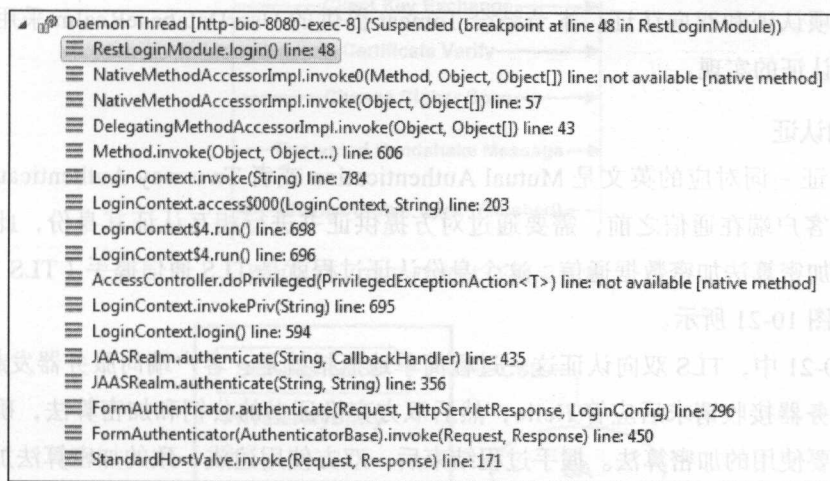


图 10-19 调用 RestLoginModule 类的 login 方法的栈

第二步，获取登录身份信息对应 RestLoginModule 类的 commit 方法，如图 10-20 所示。

到此，JAAS 在 REST 式的 Web 服务中的应用就讲述完毕，希望读者对此有清楚的认识。还要说明的是，JAAS 这种验证实现有些重型，对于轻量级的 REST 应用以及多模块分布式的 Web 服务而言，JAAS 验证的实现并不令人满意。

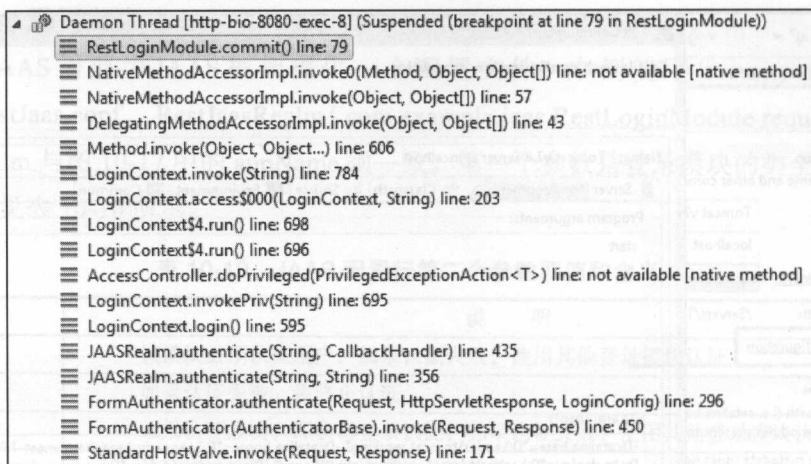


图 10-20 调用 RestLoginModule 类的 commit 方法的栈

10.3.5 证书认证与 UserDatabaseRealm

10.1 节对交代了证书认证（HTTPS）的技术背景，证书认证包括对客户端（有时是对服务器）的单项认证和双向认证。本节结合 Tomcat 提供的 UserDatabaseRealm 采用双向认证来演示证书认证的实现。

1. 双向认证

双向认证一词对应的英文是 Mutual Authentication 或者 Two way Authentication，它是指服务器和客户端在通信之前，需要通过对方提供证书进行相互认证其身份，此后双方使用协商好的加密算法加密数据通信。这个身份认证过程就是 TLS 通信握手（TLS Handshak）的过程，如图 10-21 所示。

在图 10-21 中，TLS 双向认证这一过程简单地概括就是：客户端向服务器发起 Hello 请求会话，服务器接收请求后应答 Hello，然后双方交换证书的公钥和加密算法，确认对方身份并协商将要使用的加密算法。握手过程结束后，双方使用达成一致的加密算法加密数据通信。接下来，讲述握手过程中使用的证书是如何生成并被签发和授信的。

2. 证书管理

在 Java 平台，证书（密钥信息）存储在 keystore 文件中，如图 10-22 所示。

通常的流程是系统首先在 keystore 中生成自签证书，然后将其导出为自签证书文件，接着向 CA 机构提交 CSR（Certificate Signing Request，证书签发请求）。CA 接收请求后，使用私钥签发该证书，并将 CA 签发的证书和 CA 证书（包含 CA 公钥）返回。系统将 CA 证书导入 keystore，即信任 CA——此后 TLS 通信过程中，凡是该 CA 签发的证书，该系统

皆信任。CA 签发的证书在该系统与外系统通信时，作为该系统身份认证证书提交。

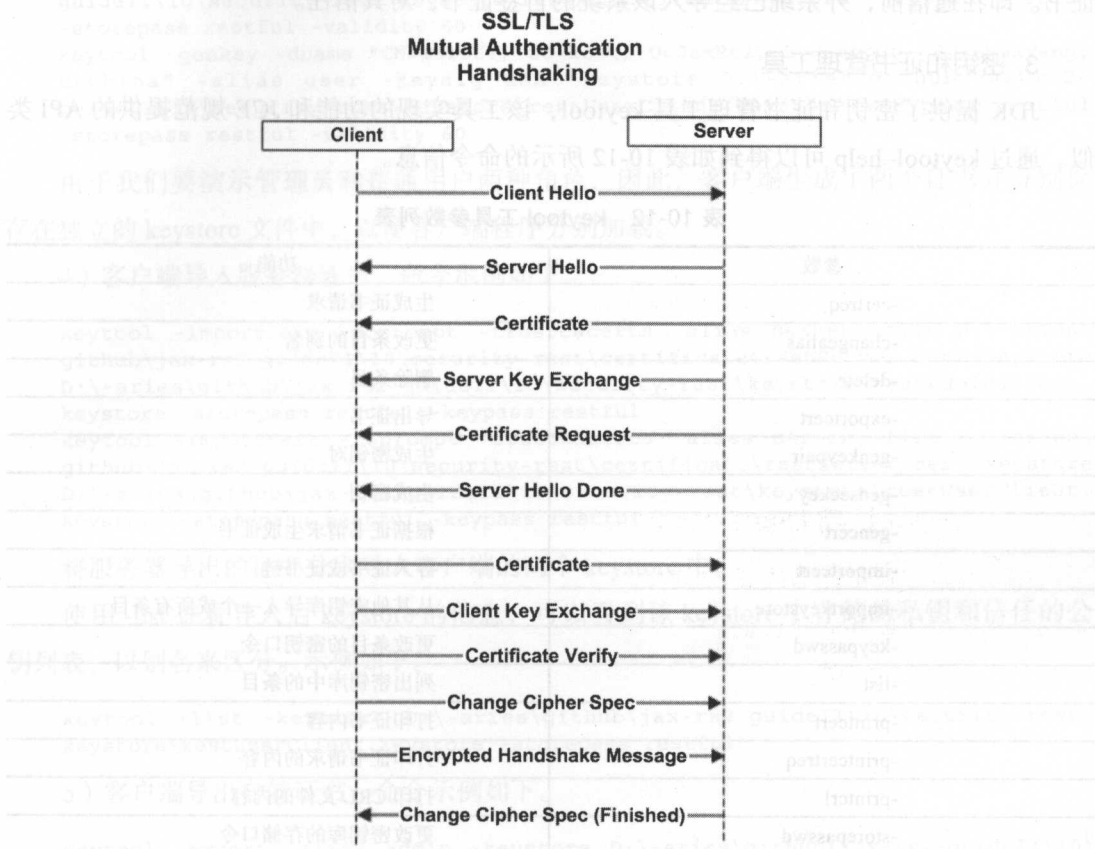


图 10-21 TLS 双向认证握手流程

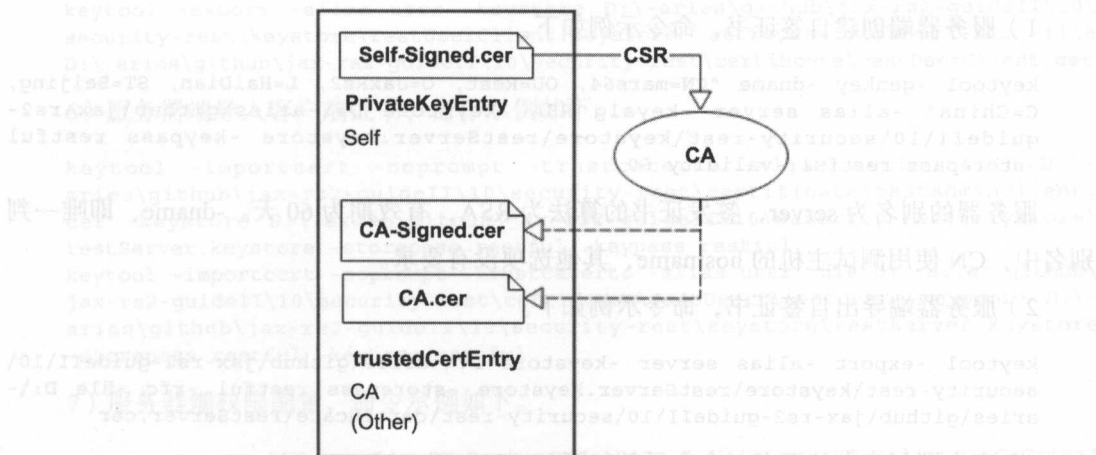


图 10-22 证书签发流程

简化的流程是省略 CA 签发证书这个流程，双方的自签证书可以作为通信时身份认证的证书。即在通信前，外系统已经导入该系统的自签证书，对其信任。

3. 密钥和证书管理工具

JDK 提供了密钥和证书管理工具 keytool，该工具实现的功能和 JCE 规范提供的 API 类似。通过 keytool-help 可以得到如表 10-12 所示的命令信息。

表 10-12 keytool 工具参数列表

| 参数 | 功能 |
|-----------------|-----------------|
| -certreq | 生成证书请求 |
| -changealias | 更改条目的别名 |
| -delete | 删除条目 |
| -exportcert | 导出证书 |
| -genkeypair | 生成密钥对 |
| -genseckey | 生成密钥 |
| -gencert | 根据证书请求生成证书 |
| -importcert | 导入证书或证书链 |
| -importkeystore | 从其他密钥库导入一个或所有条目 |
| -keypasswd | 更改条目的密钥口令 |
| -list | 列出密钥库中的条目 |
| -printcert | 打印证书内容 |
| -printcertreq | 打印证书请求的内容 |
| -printerl | 打印 CRL 文件的内容 |
| -storepasswd | 更改密钥库的存储口令 |

通过 keytool，我们可以完成上述证书管理流程。

1) 服务器端创建自签证书，命令示例如下。

```
keytool -genkey -dname "CN=mars64, OU=Rest, O=JaxRs2, L=HaiDian, ST=Beijing, C=China" -alias server -keyalg RSA -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore -keypass restful -storepass restful -validity 60
```

服务器的别名为 server，签发证书的算法为 RSA，有效期为 60 天。-dname，即唯一判别名中，CN 使用测试主机的 hostname，其他选项没有要求。

2) 服务器端导出自签证书，命令示例如下。

```
keytool -export -alias server -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore -storepass restful -rfc -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restServer.cer
```

3) 客户端创建自签证书，命令示例如下。

```
keytool -genkey -dname "CN=mars64, OU=Rest, O=JaxRs2, L=HaiDian, ST=Beijing, C=China" -alias admin -keyalg RSA -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restAdminClient.keystore -keypass restful -storepass restful -validity 60
keytool -genkey -dname "CN=mars64, OU=Rest, O=JaxRs2, L=HuangGu, ST=Shenyang, C=China" -alias user -keyalg RSA -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restUserClient.keystore -keypass restful -storepass restful -validity 60
```

由于我们要演示管理员和普通用户两种角色，因此，客户端生成了两个证书并分别保存在独立的 keystore 文件中，以便客户端程序分别加载。

4) 客户端导入服务器证书，命令示例如下。

```
keytool -importcert -noprompt -trustcacerts -alias server -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restServer.cer -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restAdminClient.keystore -storepass restful -keypass restful
keytool -importcert -noprompt -trustcacerts -alias server -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restServer.cer -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restUserClient.keystore -storepass restful -keypass restful
```

将服务器导出的证书分别导入客户端的两个 keystore 中。

使用 -list 查看导入后 keystore 的信息，可以看到该 keystore 中存储的私钥和信任的公钥列表，以别名来区分。示例如下。

```
keytool -list -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restUserClient.keystore -storepass restful
```

5) 客户端导出自签证书，命令示例如下。

```
keytool -export -alias admin -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restAdminClient.keystore -storepass restful -rfc -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restAdminClient.cer
keytool -export -alias user -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restUserClient.keystore -storepass restful -rfc -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restUserClient.cer
```

6) 服务器端导入客户端证书，命令示例如下。

```
keytool -importcert -noprompt -trustcacerts -alias admin -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restAdminClient.cer -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore -storepass restful -keypass restful
keytool -importcert -noprompt -trustcacerts -alias user -file D:\-aries\github\jax-rs2-guideII\10\security-rest\certificate\restUserClient.cer -keystore D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore -storepass restful -keypass restful
```

7) 服务器端权限配置，命令示例如下。

```
<user username="CN=mars64, OU=Rest, O=JaxRs2, L=HaiDian, ST=Beijing, C=China" password="null" roles="admin"/>
```



```
<user username="CN=mars64, OU=Rest, O=JaxRs2, L=HuangGu, ST=Shenyang, C=China"
password="null" roles="user"/>
```

证书管理完毕后，待服务器和客户端启动加载。接下来，需要对服务器端的权限进行配置。根据上述客户端的两个角色的唯一判别名，将它们的角色分别定义为 admin 和 user。

4. 配置服务器 SSL

Tomcat 服务器支持两种证书服务。一种是 APR (Apache Portable Runtime)，另一种是 JSSE。本例使用 JSSE 方式。

```
<Connector clientAuth="true" port="8443" minSpareThreads="5"
maxSpareThreads="75" enableLookups="true" disableUploadTimeout="true"
acceptCount="100" maxThreads="200" scheme="https" secure="true"
SSLEnabled="true"
keystoreFile=
"D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore"
keystoreType="JKS" keystorePass="restful"
truststoreFile=
"D:\-aries\github\jax-rs2-guideII\10\security-rest\keystore\restServer.keystore"
truststoreType="JKS" truststorePass="restful" SSLVerifyClient="require"
SSLEngine="on" sslProtocol="TLS" />
```

5. 配置应用

本例的 web.xml 配置参考前述示例，唯一不同之处是登录配置，如下所示。

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

6. 认证和授权测试

由于浏览器动态加载证书提交请求并不常见，本示例的测试参见下节的客户端实现。

10.4 JAX-RS2 实现

10.3 节通过 5 个实例展示了单独使用容器进行认证和授权的过程。本节将展示结合上述的容器认证，通过编码进行授权的流程。

10.4.1 Application 类

Application 类是容器中 REST 应用的入口，安全相关的配置要在这里完成注册，示例如下。

```
@ApplicationPath("/webapi/*")
public class AirResourceConfig extends ResourceConfig {
    public AirResourceConfig() {
```

```

    super(RolesAllowedDynamicFeature.class, BookResource.class);
}
}

```

在这段代码中，Application 类 AirResourceConfig 注册了 RolesAllowedDynamicFeature 以支持 10.2 节讲述的 JSR250 注解，还注册了资源类 BookResource 以实现 REST 服务。

10.4.2 资源类

本例中，与授权相关的资源类 BookResource 的实现与其他章节相比加入了 JSR250 的注解和 JAX-RS2 的 SecurityContext 接口，示例如下。

```

@Path("books")
public class BookResource {
    @RolesAllowed(value={"admin"})// 关注点 1: 根据角色授权访问
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Books getBooks(@Context final SecurityContext sc) {
        logMe(sc); // 关注点 2: 在方法中获取安全上下文
        final Books books = bookService.getBooks();
        BookResource.LOGGER.debug(books);
        return books;
    }
    private void logMe(final SecurityContext sc) {
        try {
            BookResource.LOGGER.info("User=" + sc.getUserPrincipal().getName());
            BookResource.LOGGER.info("User Role?=" + sc.isUserInRole("user"));
            BookResource.LOGGER.info("Auth way=" + sc.getAuthenticationScheme());
        } catch (final Exception e) {
            LOGGER.debug("Cannot print credential info."+e);
        }
    }
}

```

在这段代码中，GET 方法 getBooks 使用了 @RolesAllowed (value={"admin"}) 注解，授权访问此方法的角色为 admin，见关注点 1。此时，虽然容器的配置中定义了 user 角色可以访问这个 GET 方法，但是应用级别的权限优先级高于容器级别，在此定义的具有更细粒度的权限约束致使 user 角色不能访问 getBooks 方法。getBooks 方法定义了 SecurityContext 实例作为参数，SecurityContext 实例携带了用户角色的相关信息，运行时可以从中获取当前 getBooks 方法访问者的角色信息，见关注点 2。输出结果如下。

```

User=eric
User Role?=false
Auth way=BASIC

```

10.4.3 资源测试类

Jersey2.x 提供了基本认证连接过滤器 HttpBasicAuthFilter，摘要认证连接过滤器

HttpDigestAuthFilter, 证书认证是通过设置 JerseyClient 内部的 SSLContext 字段实现的。Jersey2.5 开始将 HttpBasicAuthFilter 和 HttpDigestAuthFilter 标记为 deprecated, 使用 HttpAuthenticationFeature 统一配置认证。

1. 基本认证测试

对于基本认证的客户端测试, 分别测试普通连接和注册 HttpBasicAuthFilter 或者 HttpAuthenticationFeature 的连接。没有提供基本认证的连接会得到服务器返回的 HTTP 状态码 401 没有通过认证的错误, 对应的预计异常类是 NotAuthorizedException。在 ClientConfig 实例中注册 HttpBasicAuthFilter 实例的连接会通过认证并取得访问 GET 方法的权限, 从服务器获得 Books 类型的数据, 示例代码如下。

```
@Test(expected = javax.ws.rs.NotAuthorizedException.class)
public void testGetAll() {
    final ClientConfig cc = new ClientConfig();
    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
}

@Test
public void testGetAll2() {
    final ClientConfig cc = new ClientConfig();
    //Jersey2.5-: cc.register(new HttpBasicAuthFilter("caroline", "zhang"));
    HttpAuthenticationFeature feature =
    HttpAuthenticationFeature.basicBuilder().nonPreemptive()
    .credentials("caroline", "zhang").build();
    cc.register(feature);
    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
}
```

2. 摘要认证测试

摘要认证的客户端测试和基本认证测试过程类似。对于基本认证的客户端测试, 分别测试普通连接和注册 HttpDigestAuthFilter 或者 HttpAuthenticationFeature 的连接。没有提供摘要认证的连接会得到服务器返回的 HTTP 状态码 401 没有通过认证的错误, 对应的预计异常类是 NotAuthorizedException。在 ClientConfig 实例中注册 HttpDigestAuthFilter 实例的连接会通过认证并取得访问 GET 方法的权限, 从服务器获得 Books 类型的数据, 示例代码如下。

```
@Test
public void testGetAll2() {
    final ClientConfig cc = new ClientConfig();
    //2.5-: cc.register(new HttpDigestAuthFilter("caroline", "zhang"));
    HttpAuthenticationFeature feature =
    HttpAuthenticationFeature.digest("caroline", "zhang");
    cc.register(feature);
}
```

```

    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
}

```

3. 证书认证测试

对于证书认证的客户端测试，需要用到 10.3.5 节证书认证的知识。首先设置 keystore 到 SslConfigurator 类的实例中，SslConfigurator 实例通过本地证书信息生成 javax.net.ssl.SSLContext 类的实例，接着 SSLContext 实例赋值于 REST 客户端作为 SSL 上下文字段，此时 REST 客户端向服务器发起握手流程既可实现上节所述的双向认证。授权逻辑请参考本章前述，示例代码如下。

```

private Client buildSecureClient(boolean admin) {
    String keystore;
    if (admin) {
        keystore = "restAdminClient.keystore";
    } else {
        keystore = "restUserClient.keystore";
    }
    final SslConfigurator sslConfig = SslConfigurator.newInstance().
trustStoreFile(keystore)
.trustStorePassword("restful").keyStoreFile(keystore).keyPassword("restful");
    final SSLContext sslContext = sslConfig.createSSLContext();
    final Client client = ClientBuilder.newBuilder().sslContext(sslContext).build();
    return client;
}

@Test
public void testGetAll() {
    final Client client = buildSecureClient(false);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
}

@Test(expected = javax.ws.rs.ForbiddenException.class)
public void testPost() {
    ...
    buildSecureClient(false).target(BASE_URI)
.request(MediaType.APPLICATION_JSON_TYPE).post(bookEntity, Book.class);
}

```

10.5 REST 服务与 OAuth2

社交网络的发展，从数据层面加速了清洗、挖掘、处理的进程，从通信层面推动了消息、认证、授权的发展。OAuth 1.0 规范 (<http://tools.ietf.org/html/rfc5849>) 和 OAuth 2.0 规范 (<http://tools.ietf.org/html/rfc6749>) 是近些年非常常见的基于社交网络的授权机制。OAuth2 目前得到更广泛的应用，它并不兼容 OAuth1，使用令牌 (Token) 代替了用户名和密码。值得注意的是，OAuth2 是一种授权协议而不是认证协议。本节将结合 RESTful 服务，

讲述 OAuth2 的服务端和客户端实践。

阅读指南

Jersey 提供了对 OAuth 服务器端和客户端的支持，如果读者的 REST 项目需要实现 OAuth，请在项目中添加如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.security</groupId>
  <artifactId>oauth1-server</artifactId>
  <version>${jersey.version}</version>
</dependency>

<dependency>
  <groupId>org.glassfish.jersey.security</groupId>
  <artifactId>oauth1-client</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

10.5.1 OAuth2 概述

在进入实战前，我们需要了解 OAuth2 的角色和不同模式，以及常用模式下的流程。

1. 角色

OAuth2 协议的通信各方的角色如下。

- ❑ 资源所有者 (Resource Owner)：访问受限资源的实体，也是被授权的实体。
- ❑ 用户代理 (User Agent)：本文中就是指浏览器。
- ❑ 第三方应用 (Client Application)：以资源所有者行为向资源服务器发起请求的应用，是 OAuth2 的客户端。
- ❑ 资源服务器 (Resource server)：受限资源的持有者。
- ❑ 授权服务器 (Authorization server)：为第三方订阅令牌 (access_token)，包括短生命周期令牌 (Access Token) 和长生命周期的令牌 (Refresh Token)。

2. 授权模式

OAuth2 的授权模式包括如下 4 种。

- ❑ 授权码模式 (Authorization Code)：是最完整、最严格的模式。用于 Web 应用。
- ❑ 简化模式 (Implicit)：不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了“授权码”步骤。用于移动应用。
- ❑ 密码模式 (Resource Owner Password Credentials)：用户向客户端提供自己的用户名和密码。客户端使用这些信息，向服务提供商索要授权。

❑ 客户端模式 (Client Credentials)：客户端以自己的名义，而不是以用户的名义，向服务提供商进行认证。

10.5.2 OAuth2 流程

授权码模式具备最完整的流程，也是我们本节唯一关注的模式，如图 10-23 所示。

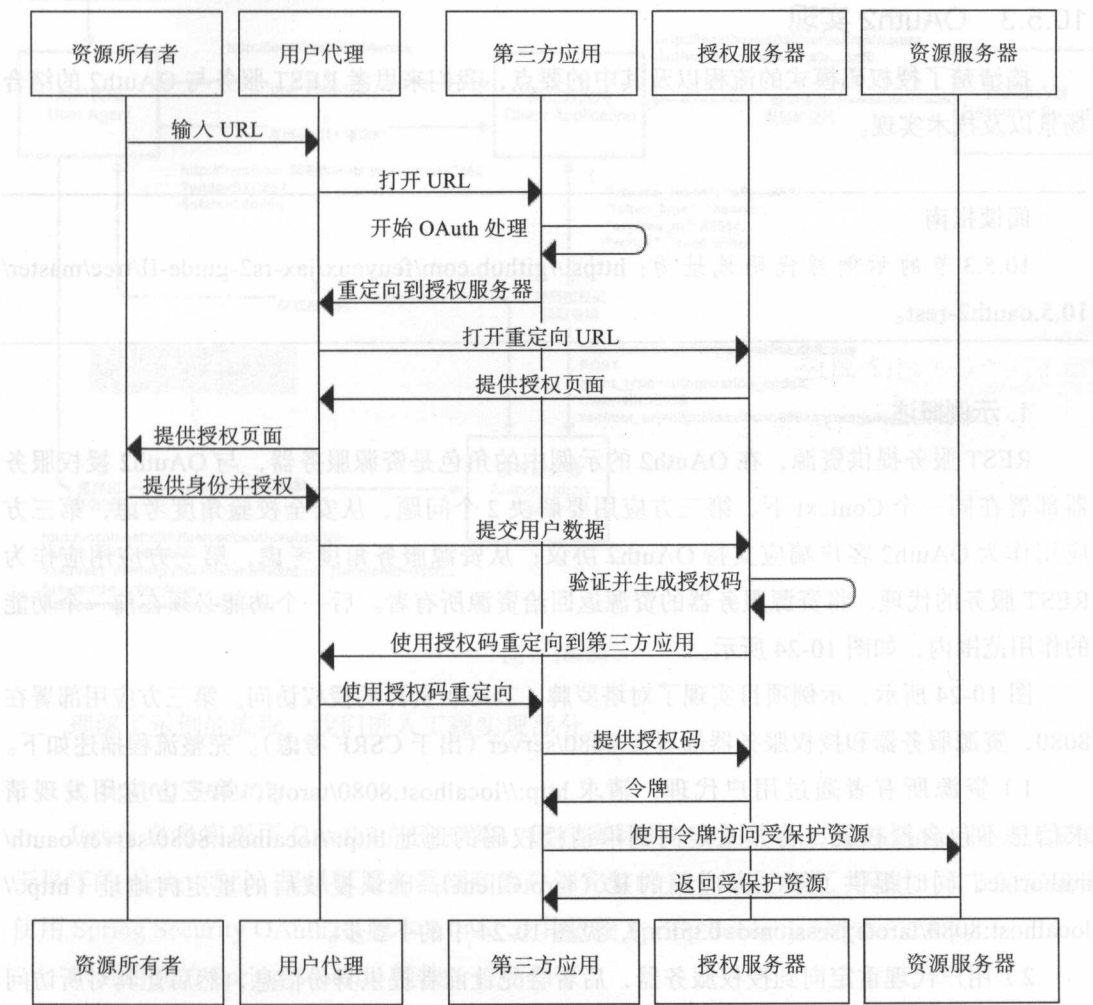


图 10-23 OAuth2 授权码模式流程

图 10-23 详细展示了 OAuth2 中各个角色的通信细节，简单来讲，图中的整个流程的目的就是资源所有者请求第三方应用，展示来自资源服务器上的资源。

流程中有如下几个关键点。

(1) 第三方应用在接受受限资源请求时, 如果没有收到授权码, 应触发 OAuth 处理流程; 否则使用授权码向授权服务器申请令牌。

(2) 资源服务器只向携带访问令牌的请求提供资源信息。

(3) 授权服务器只向认证用户同意的授权内容提供授权码。

10.5.3 OAuth2 实现

搞清楚了授权码模式的流程以及其中的要点, 我们来思考 REST 服务与 OAuth2 的结合场景以及技术实现。

阅读指南

10.5.3 节的示例源代码地址为: <https://github.com/feuyeux/jax-rs2-guide-II/tree/master/10.5.oauth2-rest>。

1. 示例概述

REST 服务提供资源, 在 OAuth2 的示例中的角色是资源服务器, 与 OAuth2 授权服务器部署在同一个 Context 下。第三方应用要解决 2 个问题、从安全校验角度考虑, 第三方应用作为 OAuth2 客户端应支持 OAuth2 协议; 从资源服务角度考虑, 第三方应用应作为 REST 服务的代理, 将资源服务器的资源返回给资源所有者。后一个功能必须在前一个功能的作用范围内, 如图 10-24 所示。

图 10-24 所示, 示例项目实现了对塔罗牌 (Tarot) 资源的授权访问。第三方应用部署在 8080, 资源服务器和授权服务器部署在 8080/server (出于 CSRF 考虑)。完整流程描述如下。

1) 资源所有者通过用户代理, 请求 <http://localhost:8080/tarots>, 第三方应用发现请求信息不包含授权码, 将其重定向到申请授权码的地址 <http://localhost:8080/server/oauth/authorize>, 同时提供了第三方应用的 ID (tarotClient), 确认授权后的重定向地址 (<http://localhost:8080/tarots;jsessionid=035E...>), 见图 10-24 中的 1-2 步。

2) 用户代理重定向到授权服务器, 后者会先让前者提供身份信息, 然后让其对所访问的资源认可访问授权, 前者认可后, 后者为其提供授权码 (BXDEiU), 重定向到第三方应用, 见图 10-24 中的 3-4 步。

3) 第三方应用发现请求信息携带授权码, 于是使用这个授权码向授权服务器地址 <http://localhost:8080/server/oauth/token> 发起 POST 请求申请访问资源的令牌, 见图 10-24 中的第 5 步。

4) 授权服务器为第三方应用提供访问令牌, 后者使用该令牌 (Authorization: bearer a5a...cdf0) 向 REST 资源服务器请求 Tarot 列表资源 (http://localhost:8080/server/rest/tarots), 见图 10-24 中的第 6 步。

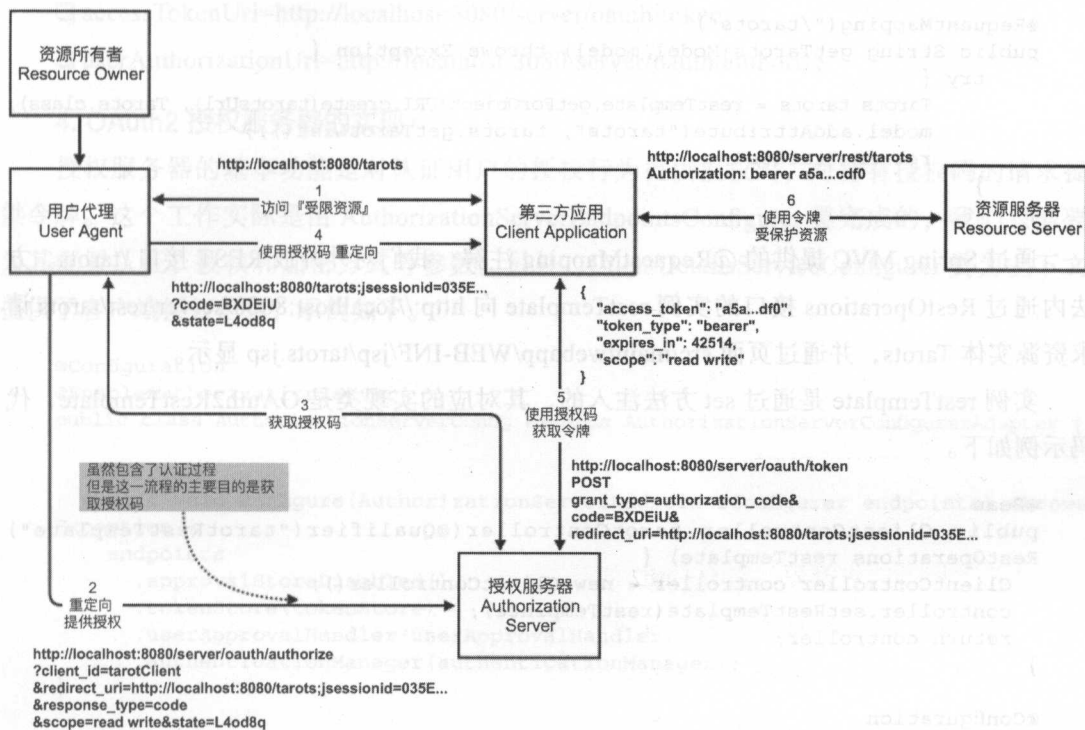


图 10-24

理解了示例的流程, 我们进入工程实现部分。

2. Spring Security

Jersey 自身实现了 OAuth2 的客户端, 但在实际的应用中, Spring Security 框架 (发展于早年的 Acegi 项目) 提供了服务器端和客户端完整的安全解决方案, 因此, 本节的实战将使用 Spring Security OAuth2 (版本 2.0.9) 及其依赖 Spring Security (版本 3.2.8)。

示例用到的 Spring Security 模块, 说明如下。

- ❑ Core: 包括认证和访问控制相关的基本功能和 API。
- ❑ Config: 解析用于配置的 Spring Security 命名空间。
- ❑ Web: 包括过滤器在内的 Web 安全相关的基础功能, 比如认证服务、基于 URL 的访问控制等。
- ❑ ACL: 标准的 ACL 领域模型实现。

3. 第三方应用的实现

第三方应用作为资源代理，对用户代理开放的 REST 服务接口必须支持 OAuth2 流程。我们首先来看资源代理服务。

```
@RequestMapping("/tarots")
public String getTarots(Model model) throws Exception {
    try {
        Tarots tarots = restTemplate.getForObject(URI.create(tarotsUrl), Tarots.class);
        model.addAttribute("tarots", tarots.getTarotList());
        return "tarots";
    }
}
```

通过 Spring MVC 提供的 `@RequestMapping` 注解，我们对外开放 REST 接口 `/tarots`。方法内通过 `RestOperations` 接口的实例 `restTemplate` 向 `http://localhost:8080/server/rest/tarots` 请求资源实体 `Tarots`，并通过页面 `src/main/webapp/WEB-INF/jsp/tarots.jsp` 显示。

实例 `restTemplate` 是通过 `set` 方法注入的，其对应的实现类是 `OAuth2RestTemplate`，代码示例如下。

```
@Bean
public ClientController tarotController(@Qualifier("tarotRestTemplate")
RestOperations restTemplate) {
    ClientController controller = new ClientController();
    controller.setRestTemplate(restTemplate);
    return controller;
}

@Configuration
@EnableOAuth2Client
protected static class OAuth2Config {
    @Bean
    public OAuth2RestTemplate tarotRestTemplate() {
        ...
        return new OAuth2RestTemplate(tarotResource(), context);
    }
}
...

```

`OAuth2RestTemplate` 构造子的第一个参数是 `OAuth2ProtectedResourceDetails` 接口，该实例构造如下。

```
@Bean
public OAuth2ProtectedResourceDetails tarotResource() {
    AuthorizationCodeResourceDetails details = new AuthorizationCodeResourceDetails();
    ...
    details.setClientId("tarotClient");
    details.setAccessTokenUri(accessTokenUri);
    details.setUserAuthorizationUri(userAuthorizationUri);
    return details;
}
```

受限资源接口的实现方式对应 OAuth2 的授权模式，我们这里使用的是授权码模式实现类 `AuthorizationCodeResourceDetails`。其中两个关键的参数是授权码申请地址和访问令牌申请地址。

❑ `accessTokenUri=http://localhost:8080/server/oauth/token`

❑ `userAuthorizationUri=http://localhost:8080/server/oauth/authorize`

4. OAuth2 授权服务器的实现

授权服务器的基本功能是对认证用户的授权行为提供授权码，对持有授权码的请求提供令牌。这个工作实际是由 `AuthorizationServerEndpointsConfigurer` 类完成的，我们只需要为其配置认证、授权和加密方式等参数。另外，`ClientDetailsServiceConfigurer` 类为上下文提供了客户端授权信息，示例如下。

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws
    Exception {
        endpoints
            .approvalStoreDisabled()
            .tokenStore(tokenStore)
            .userApprovalHandler(userApprovalHandler)
            .authenticationManager(authenticationManager);
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory().withClient("tarotClient")
            .resourceIds("tarotResourceId")
            .authorizedGrantTypes("authorization_code", "implicit")
            .authorities("ROLE_CLIENT")
    }
}
```

与认证相关的是用户的身份和角色信息、登录信息等，这个由 `WebSecurityConfigurer-Adapter` 类解决。相关配置如下。

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("han").password("han").roles(DemoConfig.ROLE)
            .and()
            .withUser("eric").password("eric").roles(DemoConfig.ROLE);
    }
}
```



```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/login.jsp").permitAll()
        .and()
        .authorizeRequests().anyRequest().hasRole(DemoConfig.ROLE)
        .and()
        .exceptionHandling().accessDeniedPage("/login.jsp?authorization_
error=true")
        .and()
        .csrf().requireCsrfProtectionMatcher(new AntPathRequestMatcher("/
oauth/authorize"))
        .disable()
        .logout().logoutSuccessUrl("/index.jsp").logoutUrl("/logout.do")
        .and()
        .formLogin().usernameParameter("j_username").passwordParameter("j_
password").failureUrl("/login.jsp?authentication_error=true").loginPage("/
login.jsp").loginProcessingUrl("/login.do");
}

```

5. 资源服务器

REST 资源服务器要解决对受限资源的保护，同时为授权请求提供相关作用域的资源开放。相关代码如下。

```

@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.resourceId("tarotResourceId");
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .requestMatchers().antMatchers("/rest/**")
            .and()
            .authorizeRequests().antMatchers("/rest/tarots").access("#oauth2.
hasScope('read')");
    }
}

```

10.6 本章小结

本章全面讲述了与 REST 安全相关的理论和实践，主要的知识点如下。

□ 身份认证

- HTTP 基本认证
- HTTP 摘要认证
- 表单认证

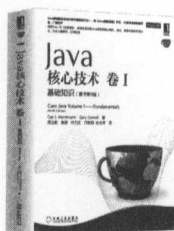
- 证书认证
- 资源授权
 - 容器管理授权
 - 应用管理授权
- 认证与授权实现
 - HTTP 基本认证 + JDBCRealm
 - HTTP 摘要认证 + UserDatabaseRealm
 - 表单认证 + DataSourceRealm
 - 表单认证 + JAASRealm
 - 证书认证 + UserDatabaseRealm
- JAX-RS2 应用管理授权
- OAuth2
 - 角色
 - 授权模式
 - 授权码模式流程
 - 授权码模式实现

另外，和其他 Java EE 标准一样，JAX-RS2 并没有给出对 XSS（Cross Site Script，跨站脚本攻击）和 CSRF（Cross-site request forgery，跨站点请求伪造）等攻击手段的防范标准。因此，一方面需要开发者在编码中防止出现代码级别的漏洞，另一方面需要开发者实现规避攻击风险的代码。举例来说，可以借助 JAX-RS2 定义 Providers 接口来实现过滤器，Jersey2.x 提供了 CSRF 防御的过滤器实现类 `CsrfProtectionFilter` 可供参考。对于 3.4 节提供的关于 CORS（Cross-Origin Resource Sharing，跨域资源共享）的实例是基于“HTTP 头不会被脚本篡改”这一理论的，在联合部署不同应用的 REST 服务的场景中，CORS 是必要的解决方案。

参考资料

- [1] Roy Thomas Fielding. 架构风格与基于网络的软件架构设计 [C]. 李锟、廖志刚、刘丹、杨光, 译. 2007.
- [2] 架构风格与基于网络应用软件的架构设计 (中文修订版). <http://www.infoq.com/cn/minibooks/web-based-apps-archit-design>. 李锟, 译.
- [3] JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services. <http://www.jcp.org/en/jsr/detail?id=339>.
- [4] Jersey 官方文档. <https://jersey.java.net/documentation/latest>.
- [5] Hypertext Transfer Protocol -- HTTP/1.1. <http://www.ietf.org/rfc/rfc2616.txt>.
- [6] URI. <http://www.ietf.org/rfc/rfc3305.txt>.
- [7] Leonard Richardson and Sam Ruby. RESTful Web Services[M]. California : O'Reilly Media, 2007.
- [8] Subbu Allamaraju. RESTful Web Services Cookbook[M]. California : O'Reilly Media, 2010.

推荐阅读



Java核心技术：卷I 基础知识（原书第9版）

作者：（美）Cay S. Horstmann Gary Cornell

译者：周立新 等

ISBN: 978-7-111-44514-2

定价：119.00元



Java核心技术：卷II 高级特性（原书第9版）

作者：（美）Cay S. Horstmann Gary Cornell

译者：陈昊鹏 等

ISBN: 978-7-111-44250-9

定价：139.00元



Java EE 7权威指南：卷1（原书第5版）

作者：（美）埃里克·珍兆科 等

译者：苏金国 等

ISBN: 978-7-111-49760-8

定价：99.00元



Java EE 7权威指南：卷2（原书第5版）

作者：（美）埃里克·珍兆科 等

译者：苏金国 等

ISBN: 978-7-111-49711-0

定价：99.00元



Java应用架构设计：模块化模式与OSGi

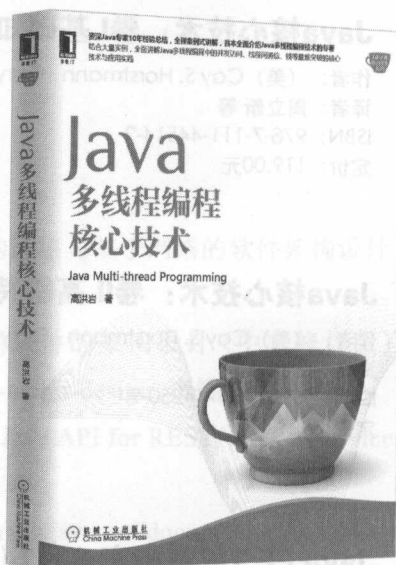
作者：（美）Kirk Knoernschild

译者：张卫滨

ISBN: 978-7-111-43768-0

定价：69.00元

推荐阅读



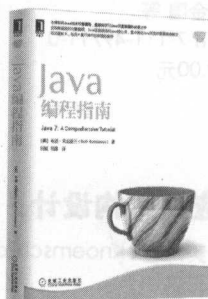
Java多线程编程核心技术

作者：高洪岩 ISBN: 978-7-111-50206-7 定价：69.00元

资深Java专家10年经验总结，全程案例式讲解，首本全面介绍Java多线程编程技术的专著。
结合大量实例，全面讲解Java多线程编程中的并发访问、线程间通信、
锁等最难突破的核心技术与应用实践。



作者：沙伦·比奥卡·扎卡沃 等
ISBN: 978-7-111-50392-7
定价：79.00元



作者：布迪·克尼亚万
ISBN: 978-7-111-50381-1
定价：99.00元



作者：蒂姆·林霍尔姆 等
ISBN: 978-7-111-50159-6
定价：79.00元

作者简介

韩 陆

北京航空航天大学软件工程硕士，Java技术专家，曾在用友（总部）、新浪、Avaya和Technicolor等知名企业从事研发工作。目前负责阿里云大数据产品的技术架构工作，实践经验非常丰富。

他是InfoQ社区编辑，利用业余时间，翻译了大量的新闻和文章，主要集中在技术架构、大数据研发、微服务、容器化、敏捷工程等领域。其中MESOS系列被InfoQ收录为电子书——《深入浅出Mesos》。

同时，他也是《JSF和Richfaces使用指南》的作者，《Java EE 7 Essentials》的译者。

Java

RESTful Web Service 实战

(第2版)

msup® | To be better 联合策划

自第1版发行后，Jersey的版本由2.9更新到了2.22.2，此间REST服务得到了更广泛的认可和使用。与此同时，Java 8、Spring Boot和Docker的爆发式发展，使得Java领域的RESTful开发有了新的发展。因此，本书第2版应运而生。

第2版部分章节在原有的基础上做了更新，新增了微服务和容器化等内容，同时删除了第1版中反馈不好的章节，旨在更精炼、更准确、更全面地阐述REST式服务，帮助读者更好地理解和应用实践。

韩陆兄是InfoQ非常优秀的社区编辑，他撰写和翻译了很多微服务、容器相关的文章，在InfoQ网站备受欢迎。从诞生到今天，REST已经有十多年的历史，并且经久不衰，被越来越多的技术团队所采用。本书是国内为数不多的系统讲解REST技术的书籍，推荐阅读。

——郭 蕾 InfoQ主编

多年前我在开发Sonatype Nexus的时候首次接触并熟悉了REST风格的Web服务，这种风格使得我们的设计简单、规范且易测，Nexus的大部分功能都可以通过一行简单的curl命令来验证，单凭这一点就足以让我喜爱上REST风格。现如今，开发REST风格的Web服务可简单多啦，尤其是如果你遵循JAX-RS 2.0标准并借助Jersey优秀的实现，编写REST风格Web服务的难度就大大降低了。本书全面且生动地阐述了JAX-RS 2.0标准，不仅覆盖了API、请求处理、安全等核心内容，更有异步通信、调优等高级主题，无疑是学习Java REST风格Web服务的绝佳参考。韩陆技术功底扎实，在写作过程中潜心阅读了大量的Jersey源码，这也让书的质量得到了进一步保障，相信本书定会助你在REST的道路上更快更稳地前行。

——许晓斌 《Maven实战》作者

投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn



上架指导: 计算机/程序设计/Java

ISBN 978-7-111-54213-



9 787111 542131

定价: 59.00元